TECHNISCHE UNIVERSITÄT DRESDEN

Deutsches Zentrum für Luft- und Raumfahrt

Fakultät Informatik Institut für Software- und Multimediatechnik, Lehrstuhl für Softwaretechnologie

Bachelor-Arbeit

Provenienz-basierte Prozessanalyse von GitLab-Projekten am Beispiel der DLR-Software BACARDI

Claas de Boer

Geboren am: 15. April 1998 in Emden
Matrikelnummer: 4604719
Immatrikulationsjahr: 2016

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

Betreuer
Dr.-Ing. Sebastian Götz (TUD)
M. Sc. Carina Haupt (DLR)
Dipl.-Math. Andreas Schreiber (DLR)

Betreuender Hochschullehrer
Prof. Dr. rer. nat. habil. Uwe Aßmann (TUD)

Eingereicht am: 3. August 2020

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit mit dem Titel *Provenienz-basierte Prozessanalyse von GitLab-Projekten am Beispiel der DLR-Software BACARDI* selbstständig und ohne unzulässige Hilfe Dritter verfasst habe. Es wurden keine anderen als die in der Arbeit angegebenen Hilfsmittel und Quellen benutzt. Die wörtlichen und sinngemäß übernommenen Zitate habe ich als solche kenntlich gemacht. Es waren keine weiteren Personen an der geistigen Herstellung der vorliegenden Arbeit beteiligt. Mir ist bekannt, dass die Nichteinhaltung dieser Erklärung zum nachträglichen Entzug des Hochschulabschlusses führen kann.

Dresden, 3. August 2020

Claas de Boer

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

1 Einleitung

1.1 Kontext der Arbeit

Die vorliegende Arbeit wurde in Kooperation mit dem Deutschen Zentrum für Luft- und Raumfahrt e.V. in der Abteilung Intelligente und Verteilte Systeme des Institutes für Softwaretechnologie verfasst.

Deutsches Zentrum für Luft- und Raumfahrt e.V.

Das Deutsche Zentrum für Luft und Raumfahrt e. V. (DLR) ist das Forschungszentrum der Bundesrepublik Deutschland für Luft- und Raumfahrt. Neben den beiden namensgebenden Themen, wird im DLR zu den Bereichen Energie, Sicherheit, Verkehr sowie Digitalisierung geforscht. Das DLR besteht aus 51 Einrichtungen und Instituten, die sich auf 27 Standorte in Deutschland und Belgien verteilen. Der Hauptsitz des DLR befindet sich in Köln. Zur Mission des DLR gehört unter anderem die Erforschung des Sonnensystems und der Erde sowie die Entwicklung nachhaltiger und umweltverträglicher Technologien. Über die eigene Forschungstätigkeit hinaus ist das DLR im Auftrag der deutschen Bundesregierung für die Planung und Umsetzung der deutschen Raumfahrtaktivitäten zuständig.

Institut für Softwaretechnologie

Zum Auftrag des Institutes für Softwaretechnologie (SC) gehört die Erforschung und Entwicklung im Gebiet der innovativen Software-Engineering-Technologien. Das Institut ist an Standorten in Köln, Braunschweig, Berlin und Oberpfaffenhofen vertreten und besteht aus den Abteilungen “Intelligente und Verteilte Systeme”, “Software für Raumfahrtsysteme und interaktive Visualisierung” sowie “High-Performance Computing”. Neben den genannten Forschungstätigkeiten stellt das Institut Softwareentwicklungs- und Softwareengineering Know-How für das gesamte DLR zur Verfügung und ist institutsübergreifend an verschiedenen Softwareprojekten beteiligt. In Kooperation mit dem DLR Institut für Raumflugbetrieb und Astronautentraining (RB) wird das Softwaresystem “Backbone Catalogue of Relational Debris Information” (BACARDI) entwickelt, dessen GitLab Projekt im Rahmen der vorliegenden Arbeit zur Evaluation verwendet wird.

1.2 Motivation

Softwareentwicklung wird zunehmend fester Bestandteil des Arbeitsalltags von Forschern und Ingenieuren - so auch im DLR. Um die Qualität der entwickelten Software zu fördern, hat das DLR im Rahmen der Software-Engineering-Initiative Workshops und Guidelines entwickelt, die Hilfestellung zur nachhaltigen Softwareentwicklung bieten. Anhand der Guidelines kann für ein konkretes Softwareprojekt ein Katalog an Maßnahmen erstellt werden, welcher Qualitäten wie die Erweiterbarkeit, Wartbarkeit und Wiederverwendbarkeit der entwickelten Software sichern soll. Die Effekte, welche die Umsetzung dieser Maßnahmen auf die Entwicklung eines Projektes haben, sind jedoch nicht einfach zu untersuchen. Projekte umfassen teils viele Entwickler sowie große Datenmengen, die über verschiedene Repositories verteilt sein können. In diesen Strukturen direkte Feedback-Schleifen zu etablieren und aufrechtzuerhalten, ist schwierig und vor allem zeitintensiv für alle beteiligten Personen. Dennoch ist es wichtig, die Effekte der Maßnahmen zu untersuchen, um künftige Maßnahmen anpassen zu können und sinnvolle von weniger sinnvollen zu unterscheiden. Eine automatisierte Methode zur Unterstützung der Prozessanalyse ist daher wünschenswert.

Im DLR wird bei der Entwicklung von Software GitLab eingesetzt. Diese Arbeit beschränkt ihre Untersuchung daher auf das Nutzungsverhalten der von einem GitLab-Projekt bereitgestellten Features.

1.3 Zielsetzung

Ziel dieser Arbeit ist es, einen Ansatz zu erarbeiten, mit Hilfe dessen die Analyse der Entwicklungsprozesse von GitLab Projekten automatisiert unterstützt werden kann. Als Grundlage der Analyse sollen dazu die Provenienzgraphen von GitLab Projekten verwendet werden. Dafür soll zunächst der Begriff der Provenienz definiert werden und die Anwendung von Provenienz auf GitLab Projekte vorgestellt werden. Zusätzlich soll auf verschiedene Graphmodelle eingegangen werden, die zur Kodierung von Provenienzgraphen in verschiedenen Serialisierungsformaten und Datenbanktypen verwendet werden. Für die persistente Speicherung und die Analyse von Provenienzgraphen über eine einheitliche Schnittstelle, soll die Graphdatenbank Neo4j verwendet werden. Für die Analyse der Provenienzgraphen der GitLab Projekte sollen Fragestellungen formuliert werden, die mit Hilfe des Provenienzgraphen eines GitLab Projektes beantwortet werden können. Zur Visualisierung der Abfrageergebnisse sollen passende Darstellungen gewählt werden.

Der erarbeitete Ansatz soll implementiert und am GitLab Projekt der DLR Software BACARDI evaluiert werden. Diese führte im Jahr 2018 einen Katalog von Maßnahmen zur Anpassung des projekteigenen Entwicklungsprozesses ein. Die Veränderungen am Entwicklungsprozess sind damit bereits formal bekannt, und die Auswirkungen auf die GitLab-Projekte kann untersucht werden. Neben den Auswirkungen von Veränderungen des Entwicklungsprozesses auf GitLab-Projekte sind auch Veränderungen am GitLab-Projekt ohne vorher bekannte Ursachen von Interesse. Bei diesen kann, ausgehend vom ermittelten Datum der Veränderung, nach der Ursache gefahndet werden.

Die vorgestellte Methode soll es zudem ermöglichen, die Provenienzdaten beliebig vieler GitLab Projekte zu vereinen, um die Untersuchung von komplexen Softwaresystemen, die aus mehreren GitLab Projekten bestehen, zu ermöglichen, ohne Anpassung an der gewählten Methodik vorzunehmen.

1.4 Aufbau der Arbeit

Im zweiten Kapitel der Arbeit wird auf die Softwareentwicklung im DLR eingegangen. Dazu wird die Software-Engineering-Initiative des DLR, der Code Hosting Service GitLab und das Forschungsfeld "Software Repository Mining" vorgestellt. Anschließend wird auf den Begriff der Provenienz und seine Anwendung auf GitLab Projekte in Form des Python Tools GitLab2PROV eingegangen. Das Kapitel schließt mit der Erläuterung des Konzeptes von Graphdatenbanken sowie der Graphdatenbank Neo4j mit ihrer Abfragesprache Cypher im Speziellen. Im dritten Kapitel, dem Methodikteil der Arbeit, wird der vierschrittige Ansatz der Analyse von Provenienzgraphen von GitLab Projekten inklusive der zur Analyse entwickelten Fragestellungen präsentiert. Die Evaluation des vorgestellten Ansatzes wird im vierten Kapitel der Arbeit neben verschiedenen GitLab Projekten am Beispiel der DLR Software BACARDI vorgenommen. Dafür wird zunächst das Softwaresystem BACARDI sowie der im BACARDI Projekt verwendete Entwicklungsprozess vorgestellt. Im fünften und letzten Kapitel wird das Fazit der Arbeit formuliert und ein Ausblick auf zukünftige Forschungsrichtungen gegeben. Der in der vorliegenden Arbeit entwickelte Ansatz wurde in einem Jupyter Notebook implementiert, welches auf GitHub[1] sowie Zenodo [Boe20] zur Verfügung steht.

[1] `https://github.com/cdboer/ba-thesis-jupyter-notebook`

2 Grundlagen

In diesem Kapitel wird das für die Arbeit nötige Hintergrundwissen vorgestellt. Dafür wird zunächst ein Einblick in die verschiedenen Aspekte der Softwareentwicklung im DLR gegeben (Abschnitt 2.1). Dabei wird darauf eingegangen, welche Maßnahmen im DLR ergriffen werden, um die Qualität der im eigenen Haus entwickelten Software zu fördern (Abschnitt 2.1.1) und welche Infrastruktur im DLR zur Softwareentwicklung genutzt wird (Abschnitt 2.1.2). Anschließend wird das Forschungsfeld des *Software Repository Minings* vorgestellt (Abschnitt 2.2), welchem diese Arbeit zuzuordnen ist. Darauf folgend wird der Begriff der *Provenienz* erörtert und definiert (Abschnitt 2.3). Dazu wird der W3C PROV Standard sowie dessen Datenmodell PROV-DM zur Modellierung von Provenienz erläutert. Im darauf folgenden Abschnitt 2.4 wird das im DLR entwickelte Python-Tool GitLab2PROV eingeführt, welches zur Generierung der in der Arbeit verwendeten Datensätze eingesetzt wird. Zum Abschluss des Grundlagenkapitels erfolgt eine Auseinandersetzung mit der Graphdatenbank Neo4j (Abschnitt 2.5.1) sowie der Graph Query Language Cypher (Abschnitt 2.5.2).

2.1 Softwareentwicklung im DLR

Das Deutsche Zentrum für Luft- und Raumfahrt (DLR) ist das Forschungszentrum der Bundesrepublik Deutschland für Luftfahrt, Raumfahrt, Energie, Verkehr und Sicherheit. Softwareentwicklung gewinnt in den Forschungstätigkeiten des DLR zunehmend an Bedeutung. Bereits über 25% der Personalkosten, die im DLR anfallen, werden für die Entwicklung von Software ausgegeben [HSM18]. Die unterschiedlichen Forschungsfelder spiegeln sich in der Diversität der entwickelten Software wieder. Unter Verwendung von Sprachen und Frameworks, wie z. B. Python, R, Perl, C, C++, Fortran, MatLab, LabView, Ada, Java, .Net, wird Software in den Bereichen Simulation und Modellierung, Flight Control, Daten- und Knowledge Management, Visualisierung, Softwareanalyse und Kommunikation entwickelt.

Die Größe üblicher Entwicklerteams reicht von einer Person in kleinen Projekten bis zu 20 Personen in größeren Teams. Häufig besteht ein Entwicklerteam aus einigen Mitarbeitern sowie unterstützenden studentischen Hilfskräften.

Bei der Betrachtung der Softwareentwicklung im DLR und der Entwicklung von Forschungssoftware im Allgemeinen ist zu berücksichtigen, dass nicht alle Personen, die in ihrem Forschungsvorhaben mit Softwareentwicklung in Kontakt kommen, einen Hintergrund in der Softwareentwicklung haben oder ausgebildete Softwareingenieure sind. Um dadurch entstehende Herausforderungen zu meistern, die Qualität und Nachhaltigkeit der entwickelten

Software zu fördern, sowie essentielles Softwareengineering Know-How an Forschende des DLR zu vermitteln, wurde 2005 die Software-Engineering-Initiative (SE Initiative) im DLR ins Leben gerufen.

2.1.1 Software-Engineering-Initiative

Ziel der Software Engineering Initiative (SE Initiative) des DLR ist die Nachhaltigkeit der im DLR entwickelten Software zu fördern. Eigenschaften, die die Nachhaltigkeit von Software maßgeblich beeinflussen, sind die Wartbarkeit, die Erweiterbarkeit sowie die Wiederverwendbarkeit von Software. Im Kontext von Forschungssoftware zielt Nachhaltigkeit im Wesentlichen darauf ab, die Nachvollziehbarkeit, Verifizierbarkeit und Reproduzierbarkeit wissenschaftlicher Erkenntnisse, sowie auch die Nachnutzbarkeit der Software sicherzustellen. Bei der Entwicklung von Forschungssoftware werden diese Eigenschaften jedoch häufig nicht priorisiert. Software wird in diesem Kontext meist nur als Mittel zum Zweck gesehen, als ein Werkzeug das funktionieren muss.

Um nachhaltige Softwareentwicklung zu fördern, ist daher ein Paradigmenwechsel notwendig. Die Entwicklung von Forschungssoftware ist Teil eines kreativen Prozesses und ist in diesem Sinne ausführbares Wissen. Forschungssoftware ist als zentrales und eigenständiges Produkt der wissenschaftlichen Arbeit zu betrachten und wertzuschätzen. Um die Nachnutzbarkeit von Forschungssoftware zu gewährleisten, ist die Sicherung nachhaltiger Eigenschaften von zentraler Bedeutung.

Eine Betrachtung der Hürden und Herausforderungen, die bei diesem Paradigmenwechsel im DLR identifiziert wurden, nehmen Haupt, Schlauch und Meinel vor [HSM18]. Dabei werden drei Punkte von ihnen zusammenfassend wie folgt benannt:

1. Fehlendes Wissen: Forschenden, die keine Erfahrung in der Softwareentwicklung haben, sind gute Praktiken der Softwareentwicklung sowie die Eigenschaften nachhaltiger Software häufig nicht geläufig. Zusätzlich benötigt die Auswahl der für ein Softwareprojekt angemessenen Vorgehensweisen einen hohen Grad an Vertrautheit mit den Methoden der Softwareentwicklung sowie mit Softwareentwicklungsprozessen im Allgemeinen.
2. Fehlende Ressourcen: Forschungsprojekte, die nicht über unabhängige Langzeitförderung finanziert werden, sind an die Rahmenbedingungen ihrer zeitlich begrenzten Projektfinanzierung gebunden. Zeitlich begrenzte Finanzierung deckt die Kosten der Wartung und des Supports für Software nur für den geplanten Projektzeitraum, nicht jedoch über diesen hinaus. Für die Nachnutzbarkeit von Software ist die Wartung jedoch von essentieller Bedeutung.

   Die Befristung der Arbeitsverträge vieler Forscher trägt zudem zum Verlust von Personen, die an der Entwicklung der Software beteiligt waren, und damit dem Verlust von Expertenwissen über die Software bei. Das Betreiben nötiger Infrastruktur zur Entwicklung von Software ist gerade für Institute, die zur Informatik traditionell weniger Bezug haben, schwierig.
3. Fehlende Motivation: Forschenden, denen die Vorteile nachhaltiger Softwareentwicklung nicht geläufig sind, schreckt der zusätzliche Aufwand häufig ab. Zudem wird in vielen Fachbereichen der Erfolg von Forschenden nicht an der Qualität der von ihnen entwickelten Software gemessen. Die Anzahl der Zitationen ihrer Veröffentlichungen oder der Reputationswert der Journals, in denen sie veröffentlichen, sind in der Bewertung des Erfolgs wichtiger. Von Seiten des Managements wird häufig die Komplexität der zu entwickelnden Software unterschätzt. Nötige Mittel zur Softwareentwicklung

werden nicht bereitgestellt, die Nachhaltigkeit der entwickelten Software wird dadurch gefährdet.

Die SE Initiative des DLR umfasst die folgenden Maßnahmen (vgl. [HSM18; HS20]).

Direktive Die Software Engineering Direktive ist Teil der Qualitätsmanagementrichtlinie des DLR, die verpflichtend für alle Institute des DLR gilt. Die Direktive betont die Bedeutung der Entwicklung nachhaltiger Software. Sie verlangt von den einzelnen Instituten sich mit den speziellen Anforderungen der Softwareentwicklung in ihrem Forschungsbereich auseinander zu setzen, Software, die wichtig für Forschungsvorhaben ist, zu bennen, nötige Infrastruktur anzuschaffen und Forschende im Bereich der Softwareentwicklung fortzubilden. Die Vernetzung der Forschenden, die in den jeweiligen Instituten Software entwickeln, in Form von Communities, ist ein weiterer wichtiger Punkt der Direktive.

Guidelines Die Software Engineering Guidelines fassen das Verständniss des DLR von guten Software Entwicklungspraktiken zusammen. Die Guidelines dienen Forschern als Einstiegspunkt in die Softwareentwicklung (Software Development, SD) und das Software Ingenieurswesen (Software Engineering, SE). Sie stellen verschiedene Themen der Bereiche wie Requirements Management, Change Management, Implementierung und Testing vor. Für jeden der Bereiche werden Handlungsempfehlungen ausgesprochen, die zusätzlich mit einer Motivation versehen sind, um die Empfehlungen nachvollziehbar zu begründen. Anhand eines Entscheidungsbaumes können Softwareprojekte in vier unterschiedliche Reifestufen eingeteilt werden. Je nach Reifestufe werden bestimmte Handlungsempfehlungen für die Entwicklung der Software verpflichtend. Um die Guidelines möglichst einfach zugänglich zu machen, sind Checklisten für die Handlungsempfehlungen der jeweiligen Reifestufen enthalten.

Fortbildungen Regelmäßige Trainings und Fortbildungen erlauben einen direkten und interaktiven Wissensaustausch.

Beratung Beratung ist Forschenden vorbehalten, die nur wenig Erfahrung im Bereich der Softwareentwicklung haben. Dazu werden Forschenden erfahrene Softwareingenieure zur Seite gestellt, die die Forschungsprojekte begleiten. Die Aufgabe der Softwareingenieure ist es, zunächst die nötige Entwicklungsinfrastruktur aufzusetzen, um anschließend zusammen mit den Forschenden, anhand der SE Guidelines, einen Entwicklungsprozess zu erarbeiten und diesen bis zum Abschluss des Forschungsprojektes zu begleiten.

Infrastruktur Um den Overhead zur nachhaltigen Softwareentwicklung für die Forschenden möglichst zu minimieren, betreibt die IT des DLR eine zentrale Infrastruktur, die allen Forschenden zur Softwareentwicklung zur Verfügung gestellt wird. Seit 2005 haben alle Mitarbeiter des DLR Zugriff auf die Versionsverwaltungssoftware Subversion oder auch den Issue Tracker MantisBT. Im Zuge der Modernisierung der bereitgestellten Infrastruktur, wird auf die Code Hosting Plattform GitLab umgestellt. Der Umstieg ist im Verlauf des Jahres 2020 geplant. Erste Institute können GitLab bereits nutzen.

Die SE Initiative ist für diese Arbeit in zweierlei Hinsicht relevant. Zum einen bieten die SE Guidelines Forschenden einen Einstieg in die Themen des Software Engineerings (SE) und des Software Developments (SD). Forscherteams werden dazu aufgefordert, die Entwicklungsprozesse ihrer Softwareprojekte an den Empfehlungen der Guidelines zu messen und gegebenenfalls anzupassen. Die Veränderungen der Entwicklungsprozesse zu untersuchen, ist daher von großem Interesse, um die Auswirkungen der Guidelines sichtbar zu machen und Rückmeldung für ihre Weiterentwicklung zu bekommen.

Zum anderen existiert mit der Einführung von GitLab als zentrale Infrastruktur zur Softwareentwicklung nun eine Möglichkeit von einer einheitlichen Schnittstelle, der GitLab API,

Metadaten und Artefakte, welche Zeugnisse des Entwicklungsprozesses darstellen, zu sammeln und anschließend zu analysieren. Dadurch wird es möglich, minimalinvasiv Einblick in den Entwicklungsprozess eines Projektes zu nehmen. Welche Features GitLab zur Verfügung stellt, und worum es sich bei GitLab handelt, wird im folgenden Absatz näher erläutert.

2.1.2 GitLab

GitLab ist eine Web basierte Code Hosting Plattform zur Begleitung und zum Management der Softwareentwicklung durch alle Stadien des DevOps Lebenszyklus [Git20]. Dafür bietet GitLab neben git basierter Versionskontrolle einen Issue Tracker, Features zum Code Review, Change Management durch Merge Requests, eine Pipeline zur Einbindung von Continuos Integration sowie Continuos Deployment, Kanban Boards und weitere Features an. Diese Features können im Rahmen eines sogenannten Projektes zur Softwareentwicklung genutzt werden. Ein GitLab Projekt wrapped ein git Repository und bietet für den Scope des Projektes einen eigenen Issue Tracker sowie projekteigene Merge Request Verwaltung an.

Ähnlich zu der vergleichbaren Plattform GitHub, bestehen Interaktionsmöglichkeiten mit denen Entwickler sich untereinander austauschen können. Durch Kommentare und Diskussionsthreads unter Features, wie Issues oder auch Merge Requests (Pull Request), wird es Entwicklern und Projektmanagern ermöglicht die Diskussion über ein Feature direkt an das Feature anzuheften, anstatt sie in davon getrennten Kommunikationsmedien zu führen. Zusätzlich ist noch eine Großzahl an Events definiert, die Aktionen wie das Erwähnen eines Entwicklers in einer Diskussion oder auch die Verteilung der Zuständigkeit für bestimmte Issues abbilden. Entgegen einer Suite an selbstständigen Softwareentwicklungstools, vereint GitLab viele Features in einer einzigen Applikation.

GitLab wird von der GitLab Inc. unter einer Open Source Lizenz öffentlich entwickelt. Zudem können Einrichtungen GitLab auf eigener Hardware laufen lassen und sind so unabhängig vom Hosting Dritter. Über die offizielle API, wahlweise REST oder GraphQL, können GitLab Aktionen, wie das Erstellen von Projekten oder Gruppen, ferngesteuert werden und Betriebsdaten einer GitLab Instanz abgefragt werden. Letzteres wird im Rahmen dieser Arbeit zum Repository Mining genutzt, um aus Metadaten und Artefakten eines GitLab Projektes Provenienzgraphen zu erstellen und anschließend zu analysieren. Das Forschungsfeld des Software Repository Minings wird im Anschluss vorgestellt.

2.2 Software Repository Mining

Softwareentwicklung an sich ist ein hoch komplexer Prozess. Von der Planung eines Softwareprojektes bis zur letzten Zeile Source Code, die während des Supports der Software verfasst wird, umfasst Softwareentwicklung eine breite Spanne an Disziplinen, Aufgaben und Herausforderungen. Fester Bestandteil moderner Softwareentwicklungsprozesse sind Werkzeuge und Tools, die Entwickler und Projektmanager bei der Softwareentwicklung unterstützen sollen. Zu diesen gehören Versionskontrollsysteme, Issue bzw. Bug Tracker oder auch Kommunikationsdienste, die während der Entwicklung genutzt werden. Aktionen, die mit Hilfe dieser Tools getätigt werden, hinterlassen Artefakte, die die Projektentwicklung z. B. in Form von Source Code Veränderungen, Kommentaren, Issue Protokollen oder auch Email Threads dokumentieren. Eine Ansammlung von Artefakten eines Software Projektes wird auch Software Repository genannt. Das Forschungsfeld Mining Software Repositories beschäftigt sich mit der systematischen Aufbereitung und Analyse dieser Artefakte. Hassan nennt zwei zentrale Aspekte des Forschungsgebiets.

> "1. The creation of techniques to automate and improve the extraction of information from repositories.
>
> 2. The discovery and validation of novel techniques and approaches to mine important information from these repositories." [Has08]

Neben der Entwicklung von Techniken, die die Extraktion von Daten aus Repositories verbessern und vereinfachen, geht es ihm auch um die Entwicklung und Überprüfung neuer Ansätze, um aus den gewonnenen Daten wichtige Informationen zu ziehen. Eine Technik zur Modellierung der Zusammenhänge und des Kontexts der extrahierten Artefakte ist die Betrachtung ihrer Provenienz. Was unter Provenienz verstanden wird, welche Tools für die Datenbeschaffung und Analyse verwendet werden und wie aus Provenienz Graphen werden, wird in den kommenden Abschnitten erläutert.

2.3 Provenienz

Der Begriff der Provenienz findet vielseitige Verwendung. Häufig taucht der Begriff im Zusammenhang mit dem Kunstgeschäft auf. Im Kunstgeschäft wird er genutzt, um die Herkunftsgeschichte und Besitzverhältnisse von Kunstgegenständen zu beschreiben.

In dieser Arbeit wird Provenienz im Kontext von elektronischen Daten und Prozessen verwendet. Dafür wird in diesem Abschnitt eine Definition des Begriffes gegeben, auf die sich im Verlauf der Arbeit gestützt wird. Zusätzlich wird erläutert, wie Provenienz aufgezeichnet und modelliert werden kann, indem der W3C PROV Standard vorgestellt wird. Schließlich wird darauf eingegangen, wie Provenienz zur Analyse von Softwareentwicklungsprozessen eingesetzt werden kann.

2.3.1 Definition von Provenienz

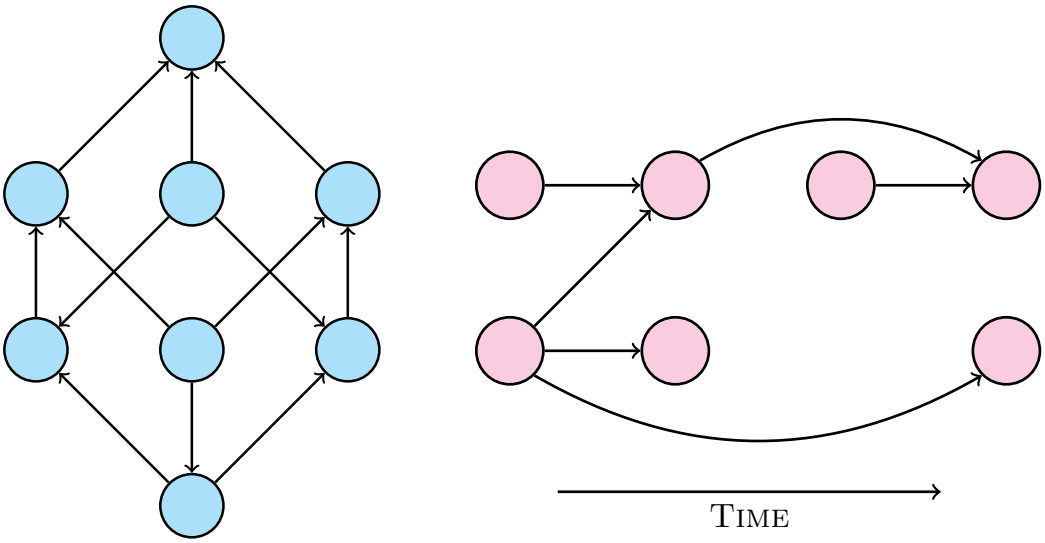
Das Konzept von Provenienz, an dem sich diese Arbeit orientiert, basiert auf der Definition von Luc Moreau.

> "The provenance of a piece of data is the process that led to that piece of data." [Mor10]

Provenienz von Daten wird verstanden, als der Prozess, der zu diesen Daten führt bzw. in diesen resultiert. In anderen Worten: Die Provenienz von Daten ist der Entstehungsprozess dieser Daten. Provenienz aufzuzeichnen lässt sich demnach auf die Aufzeichnung bzw. die Dokumentation des Entstehungsprozesses reduzieren.

Ein intuitives Modell zur Dokumentation von Prozessen ist das Aufeinanderfolgen von voneinander abhängigen Ereignissen. Dieses Modell zerlegt ein Prozessgeschehen in einzelne Teilereignisse, die miteinander in kausaler Relation stehen. Eine kausale Relation zwischen zwei Ereignissen A und B existiert genau dann, wenn A durch B herbeigeführt wird. Zwei sich bedingende Ereignisse können sich daher zeitlich nicht überschneiden. Die Ereignisse und Relationen können mit Hilfe eines gerichteten, azyklischen Graphen (DAG) dargestellt werden. Ein DAG dessen Kanten kausale Relationen repräsentieren, wird auch Kausalitätsgraph genannt. Die Kanten eines Kausalitätsgraphen verlaufen stehts von einem früheren Ereignis zu einem Ereignis, welches zu einem späteren Zeitpunkt stattfindet. Dementsprechend ist jeder Kante eine feste Richtung zugeordnet. Zusätzlich erklärt dieser Umstand die Eigenschaft der Zyklenfreiheit eines Kausalitätsgraphen. Bewegt man sich entlang eines Kantenzuges, so schreitet man stets entlang der Zeitachse von Ursache in Richtung Wirkung.

Ein Zyklus - ein Kantenzug der an dem gleichen Knoten startet an dem er auch endet - würde diese Eigenschaft verletzen. Abbildung 2.1a stellt einen gerichteten azyklischen Graph beispielhaft dar. Die Knoten des Graphen sind durch gerichtete Kanten miteinander verbunden.



(a) Gerichteter Azyklischer Graph (b) Kausalitätsgraph

Abbildung 2.1: Kausalitätsdarstellung durch gerichtete azyklische Graphen

Für die Aufzeichnung von Provenienz existieren mehrere standardisierte Ansätze [Din+10]. Die vorliegende Arbeit verwendet zu diesem Zweck den W3C PROV Standard, dessen Prozessverständnis dem eines Kausalitätsgraphen ähnelt. Die Kernkonzepte des Standards werden im folgenden Abschnitt weiter erläutert.

2.3.2 W3C PROV Standard

W3C PROV (PROV) ist ein vom World Wide Web Consortium (W3C) herausgegebener Standard zur Aufzeichnung und zum Austausch von Provenienzinformationen.

Der PROV Standard definiert Provenienz wie folgt:

> "Provenance is information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness." [Mor+13a]

W3C PROV versteht unter Provenienz den Herkunfts- beziehungsweise den Entstehungsprozess von Daten oder Dingen. Der Standard zerlegt diesen Prozess in drei verschiedene Hauptbestandteile: *Entitäten* (Entities), *Aktivitäten* (Activities), sowie *Akteure* bzw. *Agenten* (Agents). In PROV-DM, dem Datenmodell des Standards, formen diese drei Komponenten, abgebildet durch gleichnamige Typen, die Grundlagen zur Aufzeichnung von Provenienz.

PROV-DM definiert neben den Typen auch verschiedene *Relationen*, die Beziehungen der Typen zueinander abbilden. Für die Verwendung der Relationen spezifiziert der Standard ein festes Schema, das festlegt welche Typen mit Hilfe welcher Relationen miteinander verbunden werden können (Abbildung 2.2). Die Typen, sowie die für diese Arbeit relevanten Relationen, werden im Folgenden erläutert.

Entity Eine Entity ist eine physische, digitale oder anders geartete Sache, die über einige feste Eigenschaften verfügt. Dabei muss eine Entity nicht real existent sein, sondern

kann auch von imaginärer Natur sein. Entities werden, in grafischer Repräsentation, als gelb gefärbtes Oval dargestellt.

Beispiele für eine Entity sind ein Dokument, ein Himmelskörper im Universum, eine Datei in einem Dateisystem oder auch ein abstraktes Konzept, wie zum Beispiel eine Idee.

Activity Eine Activity findet über einen festen Zeitraum statt und agiert währenddessen mit oder mit Hilfe von Entities, die konsumiert, generiert, modifiziert verwendet oder verändert werden. Eine Activity bildet einen dynamischen Vorgang wie eine Aktion, ein Ereignis oder einen Prozess ab. Activities werden, in grafischer Repräsentation, als blau gefärbtes Rechteck dargestellt.

Beispiele für eine Activity sind das Editieren einer Datei, das Tätigen eines Ruderschlags, das Würfeln eines Würfels, das Veröffentlichen eines Papers, das Senden einer Kurznachricht, die Ausführung eines Skriptes oder auch das Plotten von Diagrammen.

Agent Ein Agent trägt Verantwortung für die Existenz einer Entity oder ist am Beginn und der Durchführung einer Activity beteiligt. In Verbund mit der Beteiligung an einer Activity oder der Verantwortung für eine Entity, wird dem Agent eine Rolle zugewiesen, die er in diesem Kontext wahrnimmt bzw. einnimmt. Dargestellt werden Agents als orangefarbenes Rechteck mit dreieckigem Aufsatz auf der Oberseite des Rechtecks.

Beispiele für einen Agent sind Softwaresysteme, Organisationen oder auch Mitglieder von Organisationen. Am Versenden einer Kurznachricht können gleich mehrere Agents in verschiedenen Rollen beteiligt sein. Allen voran der Verfasser der Nachricht in der Rolle des Autors, dann das Programm oder die App, die zum Versenden der Nachricht genutzt wurde, in der Rolle des Mediums. Und schließlich das Endgerät von dem die Nachricht versandt wurde, in der Rolle des verwendeten Gerätes.

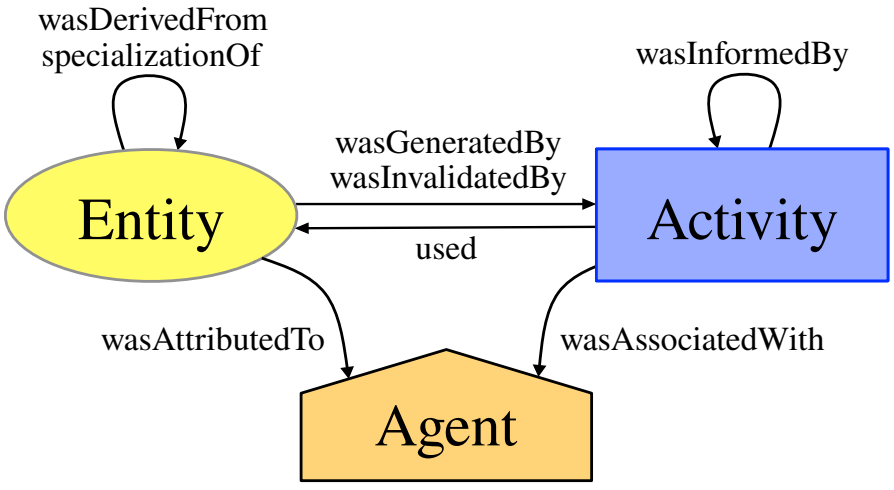


Abbildung 2.2: PROV-DM Schema - Knoten und Relationen

Die Knoten eines Provenienzgraphen sind durch semantisch benannte, gerichtete Kanten miteinander verbunden, die die Beziehungen zwischen den Knoten darstellen. Ähnlich der Kanten eines in Abschnitt 2.3.1 erläuterten Kausalitätsgraphen, haben auch in einem Provenienzgraphen die Kanten kausale Bedeutung. Eine Besonderheit ist allerdings die Richtung

der Kanten. Denn anstatt von Ursache zur Wirkung zu verlaufen, orientieren sich die Kanten in umgekehrter Richtung. So können, ausgehend von einem konkreten Knoten des Graphen, die Kanten in Aussagen über die Vorgeschichte des Knoten - dem Entstehungsprozess - verwendet werden. Der Standard definiert dafür eine Menge an Relationen sowie ein Schema, das festlegt, welche Relationen zwischen welchen Knotentypen verwendet werden dürfen. Das genannte Schema ist in Abbildung 2.2 dargestellt. Die für die vorliegende Arbeit relevanten Relationen sind:

wasGeneratedBy Wird eine Entity im Rahmen einer Activity erzeugt, so spricht man davon, dass die Entity von der Activity generiert wurde (Generation). Die Generierung findet zu einem festen Zeitpunk statt, der optional als Attribut der Relation hinzugefügt werden kann. Die Relation die eine solche Beziehung zwischen einer Entity und einer Activity repräsentiert, nennt sich *wasGeneratedBy*. Der Startknoten der Relation ist dabei stehts die generierte Entity, der Zielknoten die dafür verantwortliche Activity.

used Wird eine Entity von einer Activity verwendet und nimmt so Einfluss auf die Activity, spricht man von einer Verwendung der Entity durch die Activity (Usage). Diese Beziehung wird durch die *used* Relation dargestellt. Die Relation ist von der verwendenden Activity zur verwendeten Entity gerichtet. Wie auch bei der Generierung einer Entity, kann der Zeitpunkt des Beginns der Verwendung als Attribut der Relation ergänzt werden. Eine Activity kann mehrere Entities verwenden.

wasInformedBy Generiert eine Activity A eine Entity E, die von einer Activity B verwendet (Usage) wird, so spricht man davon, dass Kommunikation zwischen den Activities A und B besteht. Dies wird durch die *wasInformedBy* Relation dargestellt. Dabei ist der Startknoten die Activity, welche die generierte Entity verwendet. Der Zielknoten ist die Activity, die die Entity generiert. Da eine Activity beliebig viele Entities verwenden kann (vgl. used), ist auch die Anzahl der *wasInformedBy* Relationen einer Activity zu anderen Activities nicht beschränkt.

wasDerivedFrom Entsteht eine Entity durch Veränderung oder Modifizierung einer anderen, so gilt die neu enstandene Entity als Derivat (Derivation). Der PROV Standard erlaubt es Derivate in verschiedenen Detailstufen zu beschreiben. Die einfachste Stufe beschreibt eine Relation zwischen zwei Entities mit dem Namen *wasDerivedFrom*. Andere Auflösungsstufen werden in dieser Arbeit nicht betrachtet und daher, der Kürze halber, ausgelassen. Die *wasDerivedFrom* Relation verläuft von der veränderten Entity zur Unveränderten.

wasAttributedTo Eine Entity, die unter Verantwortung eines bestimmten Agents generiert wird, kann diesem zugeschrieben werden. Durch Verbindung der Entity mit dem Agent durch die *wasAttributedTo* Relation, wird dieser Sachverhalt dargestellt. Die Relation richtet sich von der generierten Entity zu dem für die Generierung verantwortlichen Agent.

wasAssociatedWith Eine Activity kann mit einem Agent assoziiert sein, sofern dieser im Rahmen einer bestimmten Rolle, Verantwortung für die Activity übernimmt. Die Assoziation der Activity mit dem Agent wird durch die *wasAssociatedWith* Relation dargestellt. Die Relation verläuft vom Knoten der Activity zum Knoten des Agent.

specializationOf Ist eine Entity eine spezielle Ausprägung einer anderen, lässt sich das durch die *specializationOf* Relation darstellen. Eine Entity ist genau dann eine spezielle Ausprägung einer anderen Entity, wenn die Entity alle Aspekte der anderen umfasst und diese um spezifische zusätzliche Informationen erweitert. Der Startknoten der Relation ist die speziell ausgeprägte Entity, der Endknoten die Entity, die durch die Ausgeprägte spezialisiert wird.

Beispielsweise ist die am Freitag bestehende Version eines Zeitungsartikels, der über die Arbeitswoche mehrere Revisionen erfuhr, eine spezielle Ausprägung des sich verändernden Artikels. Die Freitagsversion teilt alle Aspekte des sich verändernden Artikels und erweitert diese um das Datum des konkreten Wochentages sowie den Inhalt der Freitagsversion. Zwischen der Entity der Version vom Freitag und der Entity des sich verändernden Artikels kann also eine *specializationOf* Relation bestehen.

wasInvalidatedBy Wird eine Entität invalidiert, steht sie nach Invalidierung nicht mehr zur Verwendung in Aktivitäten zur Verfügung, die nach dem Zeitpunkt der Invalidierung stattfinden. Die Invalidierung stellt damit das Ende der Gültigkeit oder die Destruktion einer Entität dar. Entitäten werden von Aktivitäten invalidiert. Der Start- bzw. Ausgangsknoten der Relation ist die invalidierte Entität, der Zielknoten die invalidierende Aktivität.

Das restlose Löschen einer Datei auf einem Dateisystem ist ein Beispiel, welches sich in Form einer invalidierten Entität und einer invalidierenden Aktivität darstellen lässt. Dabei wird die Entität der Datei von der Aktivität des Löschens der Datei invalidiert.

Unter Verwendung der vorgestellten Konzepte und Relationen, ist in Abbildung 2.3 der Prozess des Pancakebackens exemplarisch als Provenienzgraph dargestellt. Zu den Entities gehören der Zucker, die Milch, das Mehl, die Eier, das Öl, die Pfanne, die erhitzte Pfanne, der Teig sowie der fertige Pancake. Die Activities sind das Vermengen der Zutaten zum Teig, das Erhitzen des Öls in der Pfanne sowie das darauf folgende Backen des Pancakes. Der im Prozess involvierte Agent, der für die Activities und damit die Generierung der Entities verantwortlich ist, ist der Koch.

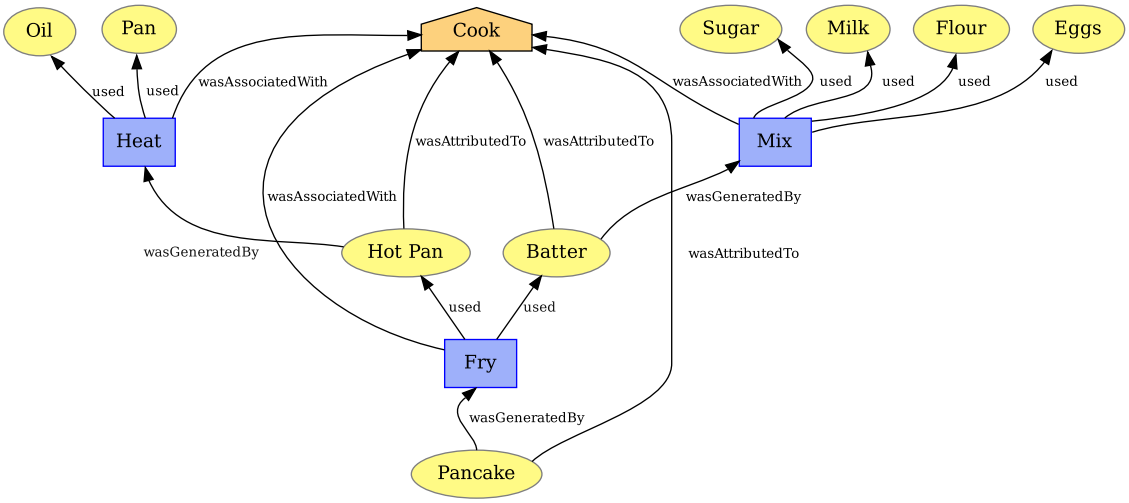


Abbildung 2.3: PROV Pancake Provenienz

Abgesehen von den bisher vorgestellten Aspekten, definiert der Standard noch einige zusätzliche Relationen und Konzepte. Darunter das Konzept der sogenannten *Bundles*. *Bundles* stellen Provenienzgraphen dar und können als Start- oder auch Endknoten von PROV Relationen, ähnlich einer Entität, verwendet werden. So lässt sich z. B. die Provenienz eines Provenienzgraphen beschreiben. Da diese im Verlauf der Arbeit nicht verwendet werden, wird an dieser Stelle nicht weiter auf sie eingegangen. Weitere Details über den Standard sowie einen Überblick über die Familie an Dokumenten die den Standard ausmacht, gibt der PROV-Overview [GM13].

Provenienz wird, wie hier eingeführt, genutzt, um den Entstehungsprozess von Daten mitsamt daran beteiligten Akteuren, Aktivitäten und Artefakten, sowie den Relationen zwischen

den einzelnen Bestandteilen, aufzuzeichnen.

2.3.3 Provenienztaxonomie

Um unterschiedliche Ansätze der Aufzeichnung und Analyse von Provenienzdaten zu klassifizieren und vergleichen zu können, wurden im Rahmen mehrerer Surveys Taxonomien erstellt [OOB18; CCM09]. In der Literatur wird zwischen zwei verschiedenen Provenienztypen unterschieden.

Prospective Provenance Prospective Provenance beschreibt den generellen Ablauf eines Prozesses und dient als eine Art Modell, Blaupause oder Rezept für den dargestellten Prozess. Provenienzmodelle, die Teile eines Prozessablaufs abstrakt modellieren, gehören zur Prospective Provenance.

Retrospective Provenance Retrospective Provenance ist die Provenienzaufzeichnung einer konkreten Prozessinstanz. Provenienzdaten, die eine konkrete Prozessinstanz wie den konkreten Ablauf eines Rechenvorganges, dokumentieren, gehören zur Retrospective Provenance.

In Abbildung 2.4 werden die verschiedenen Teilbereiche der Provenienzforschung nach der Taxonomie von Simmhan et al. dargestellt [SPG05]. Teilbereiche, denen diese Arbeit zugeschrieben werden kann, sind grün hinterlegt.

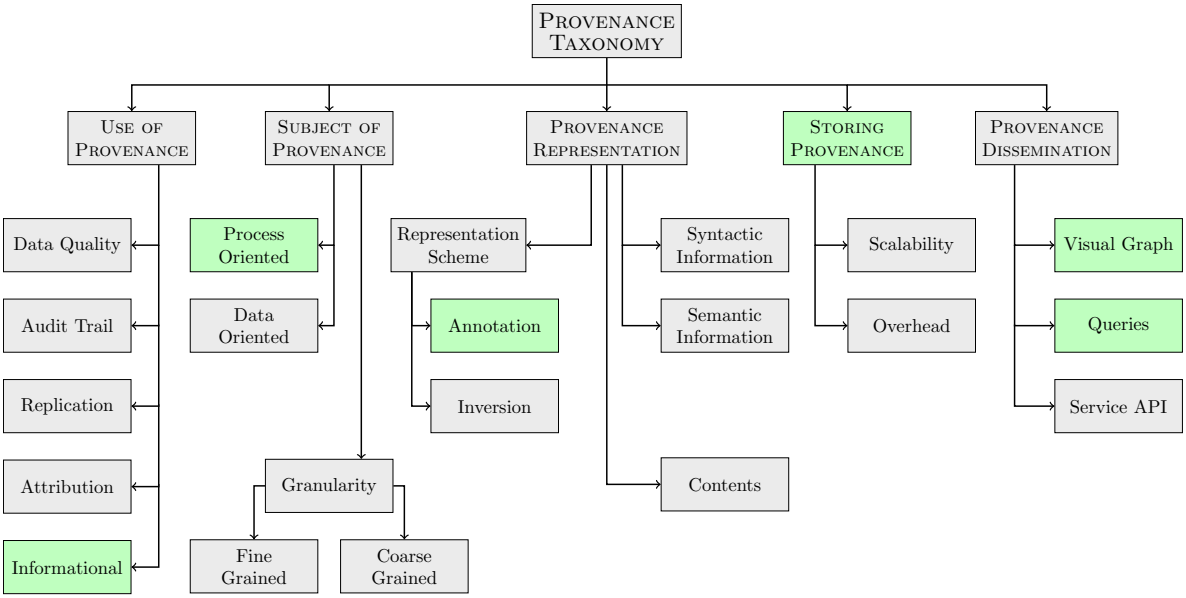


Abbildung 2.4: Provenienz Taxonomie nach Simmhan et al. [SPG05]

In Bezug auf die vorgestellten Provenienzbegriffe und die Taxonomie von Simmhan et al., kann der in dieser Arbeit verwendete Ansatz zur Aufzeichnung und Analyse von Provenienz wie folgt zusammengefasst werden: Das "Subjekt", für das in dieser Arbeit Provenienz aufgezeichnet wird, ist der Softwareentwicklungsprozess unter Nutzung des Code Hosting Services GitLab (Subject of Provenance/Process Oriented). Die aufgezeichnete Provenienz wird genutzt, um Informationen über den Entwicklungsprozess abzuleiten (Use of Provenance/Informational). Mit Hilfe eines Python Tools (GitLab2PROV) werden Projektmetadaten von GitLab Projekten über die GitLab REST-API abgefragt. Nach vordefinierten Provenienzmodellen (Prospective Provenance) rekonstruiert GitLab2PROV die Provenienzgraphen der

Projekte aus ihren Metadaten (Retrospective Provenance). Zur Aufzeichnung und Modellierung von Provenienz wird der W3C PROV Standard verwendet (Provenance Representation/Annotation). Die Provenienzdaten werden zunächst in einer RDF-JSON Datei, einem PROV Serialisierungsformat, abgelegt, bevor sie von einem Import Tool (PROV2Neo) in die Graphdatenbank Neo4j überführt werden (Storing Provenance). Zugriff auf die Daten in der Datenbank ist über die Graph Query Language Cypher möglich (Provenance Dissemination/Queries). Auszüge des gespeicherten Provenienzgraphen können im Neo4j Webinterface mittels Force-Directed Layouting [FR91] visualisiert werden (Provenance Dissemination/Visual Graph).

2.3.4 Provenienz von Softwareentwicklungsprozessen

Um die steigende Komplexität moderner Software beherrschbar zu machen, wurden neben Prozessmodellen zur Softwareentwicklung verschiedene Tools zur Unterstützung der Entwicklung entworfen. Eine typische Kollektion an Tools, die während der Entwicklung eines Softwareprojektes genutzt wird, setzt sich beispielhaft aus einem Versionskontrollsystem zum Source Code Management, einem Issue- bzw. Bug-Tracker, einem Framework zur Continuos Integration sowie Plattformen zur Applikationsdokumentation zusammen. Dokumente, Dateien, Diskussionen und Logs, die bei der Nutzung dieser Tools anfallen, dokumentieren den Entwicklungsprozess in Form von Artefakten. Der Entstehungsprozess dieser Artefakte und damit die Provenienz der Artefakte, ist Bestandteil des Entwicklungsprozesses und damit Teilmenge von dessen Provenienz.

Hosting Anbieter wie GitLab vereinen gleich mehrere der genannten Tools an einer zentralen Stelle. Prospective Provenance Modelle, wie das von Wendel et al. entwickelte [WKS10], modellieren die Beziehungen zwischen den einzelnen Artefakten, den Aktivitäten die sie generierten sowie den involvierten Akteuren beispielhaft.

2.4 GitLab2PROV

Versionskontrollsysteme (VCS) wie Git, erfassen Änderungen an Dateien bestimmter Repositories und erstellen nach jeder anfallenden Änderung neue Dateiversionen. Mit Hilfe der aufgezeichneteten Änderungen können alte Versionen wiederhergestellt werden ohne den vollständigen Inhalt der gefragten Version dauerhaft speichern zu müssen. Durch Aktionen wie Commits können Nutzer neue Dateiänderungen einem Git Repository hinzufügen.

De Nies et al. schlagen eine Abbildung des Versionskontrollsystems Git auf Modelle des Provenienzstandards PROV vor [Nie+13]. Um die Veränderungen von Dateien mit den dafür verantwortlichen Aktivitäten und Personen aufzuzeichnen, entwickeln sie mit Git2PROV Modelle für das Hinzufügen, das Modifizieren sowie das Entfernen von Dateien durch Git Commits. Mit einem von De Nies et al. entwickelten Web Service kann die Provenienz eines Git Repositories entsprechend ihrer Modelle extrahiert werden. Dazu wird das betroffene Repository temporär heruntergeladen, um durch das Parsen der Ausgabe verschiedener Git Log Befehle die nötigen Daten zur Populierung der Modelle zu sammeln. Die Daten werden anschließend in PROV-N Repräsentation gebracht, ein vom Menschen lesbares Serialisierungsformat des PROV Standards.

GitLab oder auch GitHub erweitern die Ressourcen einfacher Git Repositories um Webinterfaces und bieten darüber hinaus zusätzliche Funktionen wie Issue Tracking oder Pull bzw. Merge Request Verwaltung an. Über eine Web API können Daten zu den jeweiligen Interface

Ressourcen abgefragt werden. Ähnlich dem Entstehungsprozess neuer Dateiversionen können neue Versionen von Issues entstehen. Durch das Hinzufügen eines Kommentares oder das Vergeben eines Labels an einen Issue, entsteht eine neue Version des Issues. Diesen Ansatz verfolgen Packer et al., die die Git2PROV Modelle mit ihrem Tool GitHub2PROV um ein Modell für GitHub Issues erweitern [PCC19]. Daten, die für die Git2PROV Modelle benötigt werden, werden wie gehabt über die Ausgabe von Git Log Befehlen gesammelt. Die zur Populierung des Issue Modells nötigen Daten werden von der GitHub API abgefragt. Zusätzlich zeigen sie, dass die mit GitHub2PROV generierten Provenienzgraphen zur Beantwortung von Fragen aus dem Bereich des Projektmanagements genutzt werden können.

GitLab2PROV ist ein Python Tool zur Abbildung von GitLab Projekten auf Provenienzmodelle des PROV Standards und baut auf den von Packer und De Nies et al. verwendeten Modellen auf [BS20]. GitLab2PROV passt diese an GitLab Projekte an und erweitert sie um ein Modell für Merge Requests, sowie ein Modell für das GitLab Webinterface von Commits. Zur Beschaffung der notwendigen Daten verwendet GitLab2PROV die offizielle GitLab API. Die Abfrage der Daten wird durch Verwendung des asynchronen HTTP-Client Frameworks aiohttp [Kim19] beschleunigt. Zur Arbeit mit Provenienzdaten des PROV Standards verwendet GitLab2PROV das Python Package prov [Huy18].

Das Hinzufügen einer Datei geschieht durch einen Git Commit. Das Provenienzmodell, nach dem dieser modelliert werden kann, findet sich in Abbildung 2.5. Dieser wird durch eine entsprechende PROV Aktivität abgebildet. Den Commit verantworten zwei Agenten. Einerseits der Autor, der die im Commit verpackten Änderungen erstellte, sowie andererseits der Committer, die Person, die den Commit letztendlich durchführte. Die Commitaktivität wird daher mit beiden Agenten assoziiert. Für die hinzugefügte Datei, wird eine allgemeine Dateientität angelegt, sowie eine, die die Version wiedergibt, welche die Datei zum Zeitpunkt des Hinzufügens inne hatte. Beide Entitäten werden von der Commitaktivität generiert. Die Entität der Dateiversion stellt eine Spezialisierung der allgemeinen Dateientität dar. Zudem werden beide Entitäten dem Autor des Commits zugeschrieben, dieser ist für ihren Inhalt verantwortlich. Letzlich wird die Commitaktivität von ihrem Vorgänger, der Parent Commitaktivität, “informiert”, da sie eine Version des GitLab Projektes verwendet, die durch die Vorgängeraktivität entstand. Da ein Commit mehrere Parents haben kann, können auch mehrere Verbindungen zu verschiedenen Parentaktivitäten bestehen.

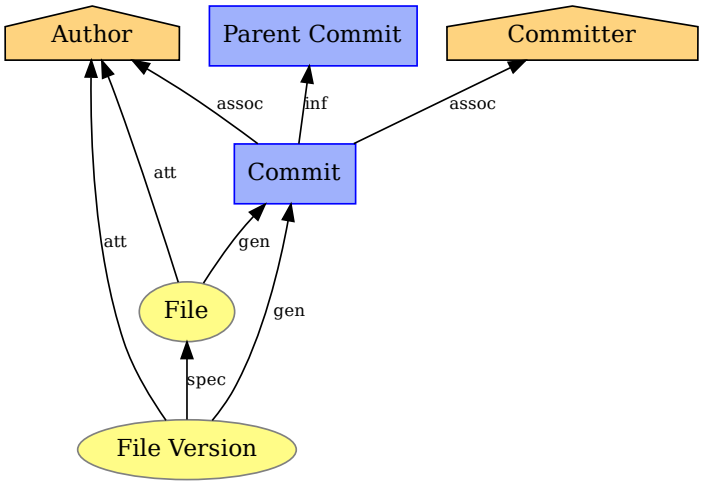


Abbildung 2.5: Git Commit Modell - Hinzufügen einer Datei durch einen Commit

Das Modifizieren einer Datei geschieht ebenfalls durch einen Commit. Das dazugehörige Provenienzmodell ist in Abbildung 2.6 dargestellt. Zu Modifizierungen gehören das Abändern des Dateiinhalts, das Umbenennen der Datei oder das Verschieben der Datei an einen anderen Speicherpfad innerhalb des Repositories. Der Commit wird durch eine PROV Aktivität abgebildet. Erneut werden zwei Agenten, der Autor sowie der Committer, mit der Commitaktivität assoziiert. Die Commitaktivität wird, wie beim Modell des Hinzufügens einer Datei durch einen Commit, mit ihrer Vorgängeraktivität, dem Parent Commit, verbunden.

Durch den Commit wird eine neue Dateiversion der veränderten Datei in Form einer Entität erzeugt. Diese wird von der Commitaktivität generiert und wird dem Autor des Commits zugeschrieben. Die Entität der neuen Dateiversion ist ein Derivat der zuletzt verfügbaren Version der Datei. Diese wird durch die "File Version N-1" Entität dargestellt. Beide Versionsentitäten, "File Version N" sowie "File Version N-1", sind wiederum spezielle Ausprägungen der allgemeinen Entität "File" der veränderten Datei, die bei dem initialen Hinzufügen der Datei erzeugt wurde.

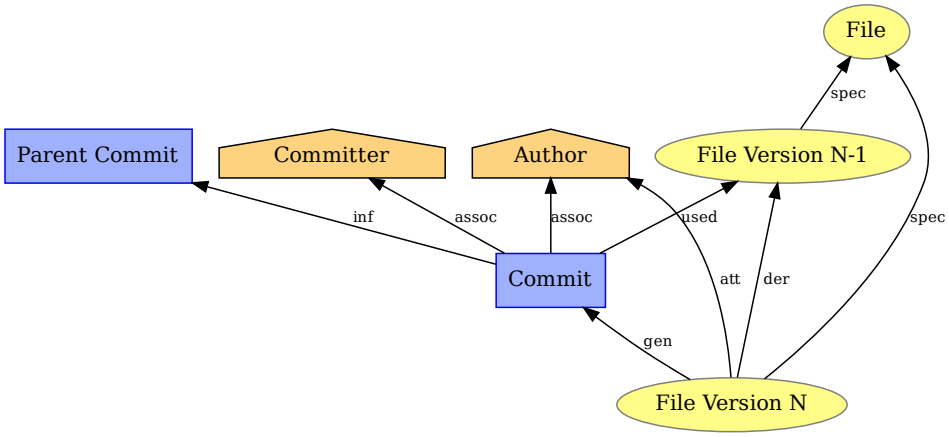


Abbildung 2.6: Git Commit Modell - Modifikation einer Datei durch einen Commit

Auch das Entfernen bzw. das Löschen einer Datei geschieht in Form eines Commits. Wie auch bei dem Hinzufügen der Datei oder dem Verändern des Dateiinhalts, wird der Commit durch eine PROV Aktivität dargestellt. Diesen Vorgang zeigt Abbildung 2.7. Erneut werden beide Personen, die am Commit beteiligt sind, durch PROV Agenten repräsentiert. Das gilt sowohl für den Autor als auch für den Committer. Auch hier wird die Commitaktivität mit ihrer Vorgängeraktivität verbunden, der Aktivität des Parent Commits. Ein Commit kann mehrere Parents haben, daher sind mehrere Verbindungen zu verschiedenen Parentaktivitäten möglich. Das Löschen einer Datei wird durch das Erzeugen einer neuen Dateiversion dargestellt, die im Gegensatz zur bisher bekannten Erzeugung von Entitäten nicht von der Commitaktivität generiert wird, sondern von dieser als ungültig markiert wird. Die als ungültig markierte Versionsentität stellt eine Ausprägung der allgemeinen Dateientität dar. Die allgemeine Dateientität wird nicht direkt als ungültig markiert, da sie in anderen Zweigen des Repositories potentiell noch gültig sein könnte.

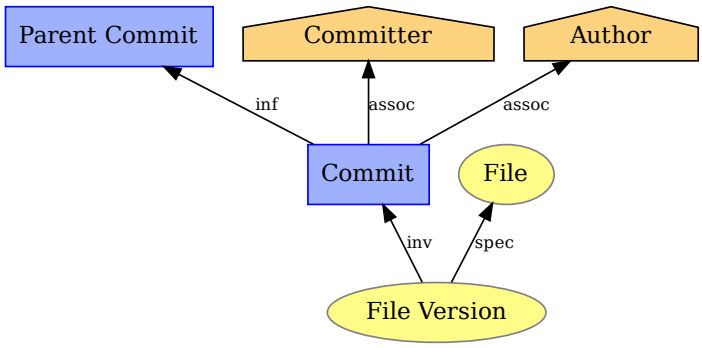


Abbildung 2.7: Git Commit Modell - Entfernen einer Datei durch einen Commit

Das in Abbildung 2.8 dargestellte GitLab Commit Modell bildet das Erzeugen des GitLab Webinterfaces für einen Commit sowie Nutzer- und Systeminteraktionen, die auf dem Webinterface stattfinden, ab. GitLab erzeugt ein Webinterface für einen Commit zum Zeitpunkt zu dem dieser dem vom GitLab Projekt gewrappten Git Repository hinzugefügt wird.

Der im Modell dargestellte Git Commit bildet eine in den Commit Modellen verwendete Aktivität ab. Das GitLab Commit Modell sowie die Git Commit Modelle sind an dieser Stelle miteinander verbunden. Das Erzeugen des Webinterfaces wird durch die Commit Creation Aktivität dargestellt. Diese erzeugt eine allgemeine Entität (Commit) zur Repräsentation des Webinterfaces sowie eine Entität (Commit Version), die die erste Version des Interfaces abbildet. Die Commit Creation Aktivität wird mit der dafür verantwortlichen Person assoziiert, dem Creator. Der Creator ist im Git Commit Modell zeitgleich Committer und damit verantwortlich für den Git Commit, für welchen das Webinterface erstellt wird. Die erzeugten Entitäten des Webinterfaces werden dem Creator Agent zugeschrieben.

Neue Versionen des Webinterfaces können durch Aktionen, die von Seiten der Nutzer oder durch GitLab durchgeführt werden, generiert werden. Ein Beispiel für eine Nutzeraktion ist das Schreiben eines Kommentares in der Kommentarsektion des Commit Webinterfaces. Wird ein Commit im Webinterface einer anderen Ressource, wie dem eines Issues oder dem einer Merge Request erwähnt, so versieht GitLab das Webinterface des entsprechenden Commits mit einem Hinweis. Aktionen dieser Art werden im weiteren Verlauf auch als Systemaktionen bezeichnet.

Eine Annotation dieser Art wird mit einer Person, dem Annotator Agent, assoziiert. Die Annotation nutzt die aktuellste vorherige Versionsentität und generiert eine neue Versionsentität, in Abbildung 2.8 die Annotated Commit Version Entity. Diese stellt eine Spezialisierung der allgemeinen Entität des Commit Webinterfaces dar und ist ein Derivat der vorherigen Version. Die Annotated Version Entität wird dem Annotator Agent zugeschrieben.

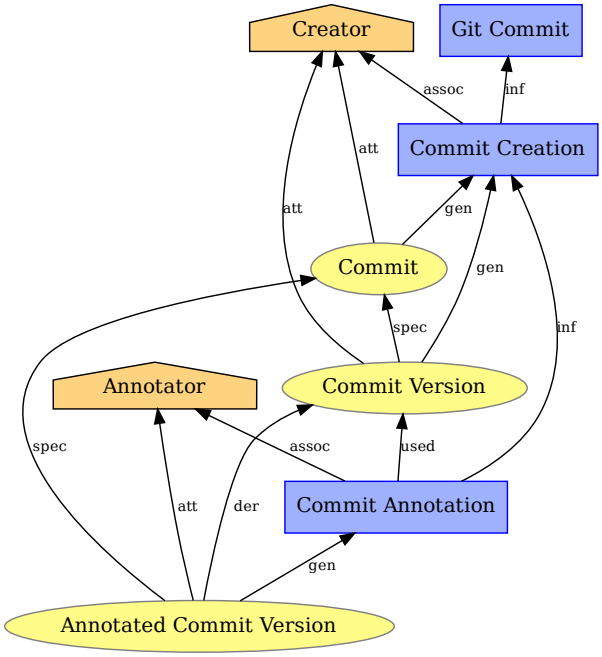
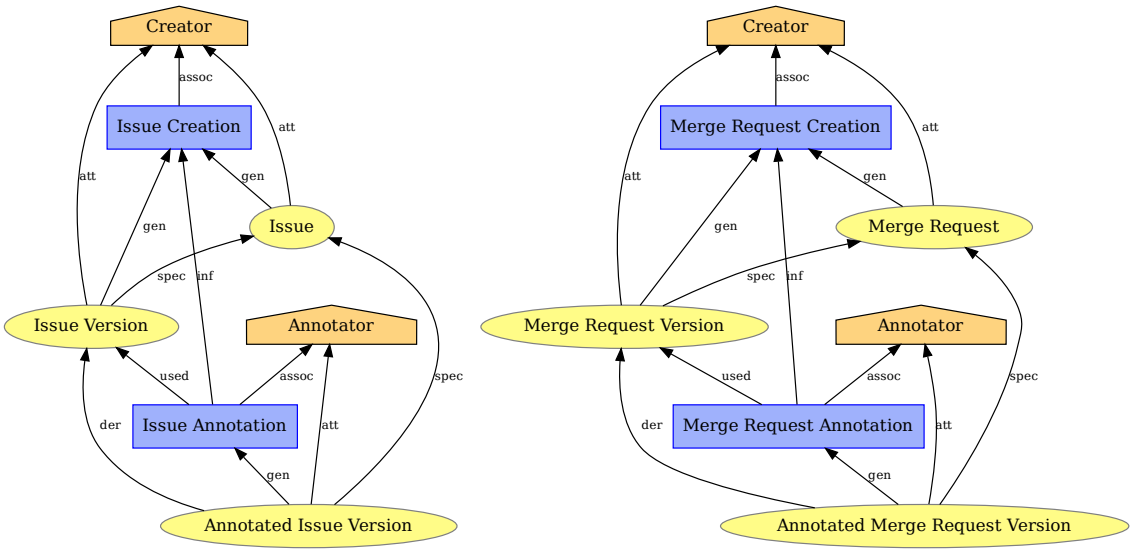


Abbildung 2.8: GitLab Commit Modell - Modellierung der Nutzer- und Systeminteraktionen eines GitLab Commit Webinterfaces

Das GitLab Issue Modell bildet, wie in Abbildung 2.9a ausgeführt, wie das GitLab Commit Modell, das Webinterface eines Issues mit dessen Nutzer- sowie Systeminteraktionen ab. Ein Teil, wie das Erzeugen einer neuen Version eines Issues, gleicht dem GitLab Commit Modell. Die initiale Generierung des Issues weicht von der Generierung eines Commit Webinterfaces ab. Im Modell wird ein Issue durch die Issue Creation Aktivität generiert. Diese ist, im Gegensatz zur Commit Creation Aktivität des GitLab Commit Modells, nicht mit den Git Commit Modellen verbunden. Bis auf dieses Detail sowie andere Bezeichnungen für Agenten, Entitäten und Aktivitäten, ist das Modell für GitLab Issues identisch zu dem des GitLab Commit Modells.

Wie Abbildung 2.9b zeigt, ist das für Merge Requests gewählte Modell bis auf Bezeichnungen der Typen sowie die Rollen der Agenten identisch zu dem des GitLab Issue Modells. Auch hier unterscheidet sich die Menge der Interaktionen, die mit Merge Requests getätigt werden können.



(a) Issue Modell

(b) Merge Request Modell

Abbildung 2.9: Modellierung der Entstehung von GitLab Issues und Merge Requests sowie Modellierung der Nutzer- und Systeminteraktionen mit den jeweiligen Ressourcen

2.5 Graphdatenbanken

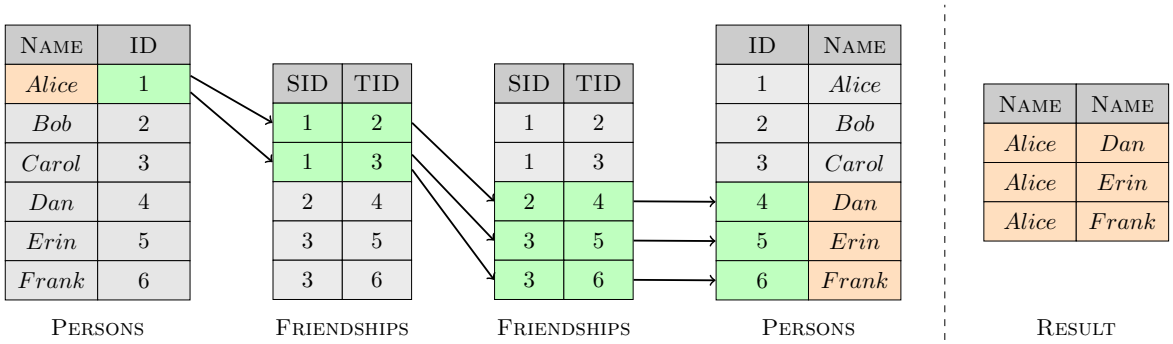
Die International Data Corporation (IDC) sagt voraus, dass die Gesamtgröße der jährlich weltweit anfallenden, elektronischen Daten von 33 Zettabyte im Jahr 2018, auf schätzungsweise 175 Zettabyte im Jahr 2025 ansteigen wird [Ryd18]. Die effiziente, persistente Haltung von Daten sowie die Bereitstellung der Daten über klare Schnittstellen, ist bereits seit den 1970ern ein wichtiger Forschungsbereich. Zu diesem Zweck, werden sogenannte Datenbankmanagementsysteme (*DMBS*) eingesetzt, die auf verschiedenen Datenmodelle beruhen.

Das am weitesten verbreitete Modell ist das der Relationalen Datenbankmanagementsysteme (*RDBMS*). In *RDBMS* werden Daten jeglicher Art in Tabellen, bestehend aus Zeilen und Spalten abgespeichert. Operationen, die auf den Tabellen ausgeführt werden können, werden durch die relationale Algebra beschrieben. Die relationale Algebra bildet die theoretische Grundlage relationaler Datenbanken. Um eine bestimmte Domäne mit Hilfe eines RDBMS abzubilden, muss ein festes Datenschema erarbeitet werden. Das Schema legt fest, wie viele Spalten eine bestimmte Tabelle haben darf, sowie welche Datentypen in den jeweiligen Spalten verwendet werden dürfen. Zum Abrufen von Informationen wird die standardisierte Abfragesprache SQL verwendet. Verschiedene Implementierungen von *RDBMS* teilen sich mit SQL die gleiche Abfragesprache. Dadurch erreicht die Nutzung von *RDBMS* hohe Portabilität, da die genutzte Implementierung einfach ausgetauscht werden kann, ohne Applikationscode anpassen zu müssen.

Aufgrund ihres festen Schemas sind *RDBMS* jedoch nicht für alle Datensätze ideal geeignet. Verändert sich die Struktur einer Domäne stark, so muss das Schema stetig angepasst werden, um die Domäne weiterhin abbilden zu können. Daraus resultiert ein hoher Wartungsaufwand.

Verbindungen zwischen Datenpunkten verschiedener Tabellen, werden mittels Primär- und Fremdschlüsseln realisiert. Ein Schlüssel ist eine Kombination von Spalten einer Tabelle, die die Zeilen einer Tabelle eindeutig identifiziert. Referenzierende Tabellen verwenden Schlüsselwerte in Form eines Fremdschlüssels. Abfragen, die mehrere Tabellen miteinander verbinden, werden mittels der sogenanntes JOIN Operation ausgeführt. Dabei werden die Primärschlüssel der Tabellen mit den referenzierenden Fremdschlüsseln abgeglichen. Mit einer Komplexität von $\mathcal{O}(\mathcal{N} * \mathcal{M})$, wobei $\mathcal{N}$ und $\mathcal{M}$ die Anzahl der Einträge der zu JOINenden Tabellen darstellen, ist diese Operation vergleichsweise rechenintensiv.

Abbildung 2.10 stellt eine Abfrage dar, die JOIN Operationen verwendet, um die Namen der Freunde von Freunden von Alice zu erfahren. In der dafür genutzten relationalen Datenbank existieren genau zwei Tabellen. Die Tabelle *Persons*, mit den Spalten Name und ID, sowie die Tabelle *Friendships* mit den Spalten SourceID und TargetID. Die Spalten SourceID und TargetID enthalten ID Werte der Tabelle Persons als Fremdschlüssel. Zeileneinträge der Tabelle kodieren so Freundschaftsbeziehungen zwischen je zwei Personen. Um die dargestellte Abfrage nach den Namen der Freunde von Freunden von Alice zu beantworten, müssen gleich drei JOIN Operationen durchgeführt werden. Die erste zwischen *Persons* und *Friendships*, um alle Personen zu erfragen, von denen eine Freundschaftbeziehung ausgeht. Die zweite zwischen *Friendships* und *Friendships*, um Freundschaftsrelationen der Länge 2 zu konstruieren. Und letztlich wieder zwischen *Friendships* und *Persons*, um die Namen der Personen zu bekommen, die über eine Freundschaftsbeziehung der Länge zwei mit anderen Personen in Relation stehen.




```
1  -- FoF := Friend of Friend; TID := targetID; SID := sourceID
2  SELECT Persons.name, FoF.name
3  FROM Persons
4  INNER JOIN Friendships AS f1  ON Persons.ID  = f1.sourceID
5  INNER JOIN Friendships AS f2  ON f1.targetID = f2.sourceID
6  INNER JOIN Persons     AS FoF ON f2.targetID = FoF.ID
7  WHERE Persons.name = "Alice";
```

Abbildung 2.10: Wie heißen die Freunde der Freunde von Alice? Beispiel einer Multi-Hop Query - einer Abfrage die Beziehungen zwischen Tabellen mehrfach traversiert.

Abfragen, die eine große Anzahl von Verbindungen zwischen Datenpunkten traversieren, müssen auch eine große Anzahl an JOIN Operationen durchführen. Das verlängert die Wartezeit auf Ergebnisse. Für die strukturelle und topologische Analyse stark vernetzter Datensätze, wie der Auswertung von Protein-Protein Interaktionsnetzwerken, der Analyse von sozialen Netzwerken oder der Provenienzanalyse, sind *RDBMS* daher nicht ideal geeignet [Vic+10].

Um verschiedene Nachteile von *RDBMS* zu überwinden, wurden Datenbanksysteme entwickelt, die weder auf dem traditionell relationalen Modell beruhen, noch SQL zur Abfrage von Daten verwenden. Datenbanksysteme dieser Art lassen sich unter dem Namen der NoSQL Datenbanken zusammenfassen. Zu der Familie der NoSQL Datenbanken zählen auch die Graphdatenbanken. Im Unterschied zu *RDBMS* speichern Graphdatenbanken Daten nativ in einem Graphmodell. Die zwei prominentesten Graphmodelle sind das Resource Description Framework und das Modell der Labeled Property Graphs.

RDF Graph Modell

Das Resource Description Framework (RDF) ist ein Standard des World Wide Web Consortiums (W3C) zum Austausch von Graphen aus dem Bereich des Semantischen Webs [WLC14]. Das Kernkonzept des RDF Standards leitet sich aus seinem Namen ab: Aussagen über Resourcen treffen bzw. Eigenschaften von Resourcen beschreiben. Resourcen und ihre Eigenschaften werden im RDF Standard durch einen gerichteten, kanten-gelabelten Multi-Graph (vlg. [RN10] dargestellt. Resourcen sowie Literale, die konkrete Werte, wie z. B. Zahlen oder Zeichenketten abbilden, stellen die Knoten des Graphen dar. Eigenschaften werden durch die Kanten des Graphen abgebildet.

Identifikatoren Als Identifikatoren einzelner Resourcen und Kanten werden IRIs (International Resource Identifier) verwendet. IRIs sind eine Erweiterung von URIs (Uniform Resource Identifier) um UTF-8 Zeichen. URIs werden häufig mit den im Web verwendeten URLs (Uniform Resource Locator) verwechselt. Tatsächlich umfasst die Menge aller URIs auch alle URLs, jedoch können neben URLs noch verschiedene andere Zeichenketten durch URIs kodiert werden.

Literale Um konkrete Werte wie Zeichenketten, Zahlen, oder Werte anderer spezieller Datentypen darzustellen, werden sogenannte Literale verwendet. Diese kodieren Werte in Form einer Zeichenkette bestehend aus zwei Teilstücken, dem lexikalischen Wert sowie dem Identifikator des Datentyps des kodierten Wertes. Seit RDF 1.1 kann für Datentypen wie Integer oder Strings der lexikalische Wert notiert werden ohne den Datentyp mit angeben zu müssen.

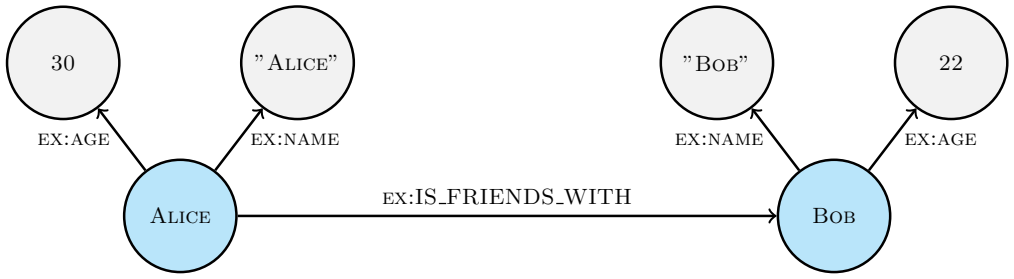
Tripel RDF kodiert Aussagen über Resourcen in Form von Tripeln. Datenbanken, die RDF Daten abspeichern, werden daher auch als Tripel Store bezeichnet. Ein Tripel ist aufgeteilt in Subjekt, Prädikat und Objekt. Das Subjekt stellt eine Resource in Form einer IRI dar. Das Prädikat bildet eine Eigenschaft der Resource in Form einer IRI sowie zugleich eine Relation zu einer weiteren Ressource, dem Objekt (in Form einer IRI oder eines Literals), ab. Das Subjekt ist also der Ausgangsknoten einer Kante, das Prädikat das Label bzw. der Identifikator der Kante und das Objekt der Zielknoten der Kante. Durch eine Menge von Tripeln lassen sich auf diese Art und Weise Graphen aufspannen. Neben IRIs von Resourcen dürfen auch IRIs von Kanten als Subjekt eines Tripels verwendet werden. Die Beschreibung der Eigenschaften einer Kante auf diese Art nennt sich “Qualifizierung”.

Quads Neben Tripeln können im RDF Standard auch Quads verwendet werden. Quads erweitern die bereits bekannten Tripel um eine weitere Stelle. Die hinzugefügte Stelle wird für einen zusätzlichen Identifikator verwendet, der die Zugehörigkeit des Tripels zu einer benannten Menge darstellt. Quads erweitern RDF um die Möglichkeit benannte Subgraphen zu beschreiben, da jede benannte Menge von Tripeln selbst einen Graphen aufspannt, der mit dem Identifikator der Menge einen Namen trägt. Der Identifikator des Graphen, der durch die benannte Menge aufgespannt wird, kann als Subjekt oder Objekt weiterer Tripel verwendet werden. Dadurch lassen sich Beziehungen zwischen Graphen oder auch Beziehungen zwischen Knoten und Graphen darstellen. Im W3C PROV Standard dienen Quads

als die Grundlage für *Bundles*.

Abfragesprache Für RDF spezifiziert das W3C die Abfragesprache SPARQL, die durch den gleichnamigen W3C Standard definiert wird [Gro13]. Mit SPARQL existiert für Daten des RDF Graphmodell eine standardisierte einheitliche Abfragesprache. Ähnlich zu RDBMS, können Tripel Store Implementierungen daher leicht ausgetauscht werden.

Für Daten des RDF Standards gibt es verschiedene Serialisierungsformate. Das im PROV Standard definierte Serialisierungsformat PROV-N [Mor+13b] baut auf diesen Formaten auf. Ein Beispiel eines RDF Graphen, der aus Tripeln besteht, sowie dessen Serialisierung in Turtle Format [Bec+14], ist in Abbildung 2.11 dargestellt. Der abgebildete Graph stellt eine, aus Abbildung 2.10 entliehene, Freundschaftsbeziehung zwischen Alice und Bob dar. Eigenschaften wie der Name oder das Alter von Alice und Bob, werden durch entsprechende Relationen abgebildet. Turtle erlaubt es IRIs mittels sogenannter Präfixe abzukürzen. Das wird im Beispiel für die IRI *http://example.org* getan. Pro Zeile wird ein Tripel notiert. Die einzelnen Teilstücke eines Tripels werden durch Leerzeichen voneinander abgetrennt. Ein Tripel wird durch einen Punkt abgeschlossen.




```
1  # 'prefix' erzeugt Kürzel 'ex:' für URI 'http://example.org'
2  @prefix  ex: <http://example.org> .
3
4  # Subjekt Prädikat Objekt
5  ex:bob ex:age 22 .
6  ex:bob ex:name "Bob" .
7  ex:alice ex:age 30 .
8  ex:alice ex:name "Alice" .
9  ex:alice ex:IS_FRIENDS_WITH ex:bob .
```

Abbildung 2.11: RDF Graph – RDF in Turtle Syntax

Ein prominentes Beispiel einer großen Datenbank auf Basis des RDF Standards mit einem öffentlichem SPARQL Query Interface ist das Wikidata Projekt der Wikimedia Foundation [VK14].

LPG Graph Modell

Labeled Property Graphen (LPG) sind gerichtete, gelabelte Multi-Graphen (vgl. [RN10]), deren Knoten und Kanten Attribute in Form von Key-Value Paaren zugeschrieben bekommen können. Abbildung 2.12 stellt die im Beispiel der SQL Query in Abbildung 2.10 verwendete Datenbasis noch einmal als Labeled Property Graph dar. Die Knoten des Graphen repräsentieren die jeweiligen Personen, die Kanten explizit die Freundschaftsrelationen zwischen den Personen. Im relationalen Modell musste dies noch implizit durch eine seperate Tabelle dargestellt werden.

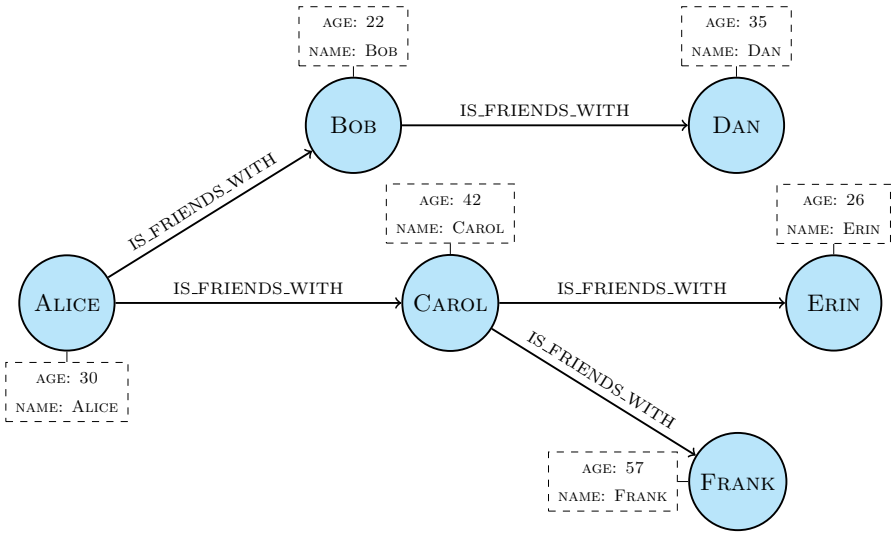


Abbildung 2.12: Property Graph Beispiel anhand von Freundschaftsbeziehungen

Knoten Im LPG Modell können Knoten eine beliebige Anzahl an Attributen, sogenannte Properties, zugeschrieben bekommen. Properties sind eine Menge von Key-Value Attributpaaren. Zudem können Knoten mit einem oder mehreren Labeln versehen werden, die den Typ oder die Rolle eines Knoten im Graphen notiert.

Kanten Kanten haben im LPG Modell stets eine feste Richtung, einen Kantentyp sowie einen Start- und Endknoten. Genau wie Knoten, können Kanten Attribute in Form von Key-Value Attributpaaren zugeschrieben bekommen. Im Gegensatz zum RDF Graph Modell werden Kanten nur zwischen je zwei Knoten zugelassen.

Abfragesprache Für Graphdatenbanken, denen das Labelled Property Graph Modell zu Grunde liegt, gibt es zum Stand dieser Arbeit noch keine einheitliche, standardisierte Abfragesprache. Mit dem Graph Query Language GQL ISO Standard ist eine standardisierte Abfragesprache in Arbeit [GQL20]. GQL soll Sprachfeature und Konzepte bisher bestehender Property Graph Query Languages wie Cypher, G-CORE, PGQL oder auch der SQL Graph Erweiterung GSQL übernehmen und darauf aufbauen. Ziel ist es unter anderem Daten im Property Graph Modell nativ auf SQL basierten Systemen unterstützen zu können.

Im Hauptteil dieser Arbeit wird zur Speicherung und Analyse von Provenienzgraphen die auf dem Property Graph Modell beruhende Graphdatenbank Neo4j verwendet.

2.5.1 Neo4j

Neo4j ist eine ACID kompatible, transaktionale Graphdatenbank, deren CRUD (Create, Read, Update, Delete) Operationen nativ auf Basis des Labeled Property Graph Modells implementiert sind [Inc20]. Neo4j ist in Java geschrieben und wird für nichtkommerzielle Anwender kostenlos als Open Source Version unter der GNU General Public License v3 zur Verfügung gestellt. Die aktuellste Version zum Stand dieser Arbeit ist der Release der Version 4.0.5. Über eine integrierte REST-Web Schnittstelle oder wahlweise über das Binärprotokoll Bolt, können Nutzer mit einer Neo4j Instanz interagieren, Abfragen stellen oder Daten integrieren. Neo4j wird im Hauptteil dieser Arbeit zur Speicherung und Analyse von Provenienzgraphen genutzt.

Neo4j speichert Knoten in einem Index zusammen mit einer Liste der vom Knoten ausgehenden Relationen. Im Gegensatz zu relationalen Datenbanken, werden die Relationen

zwischen Knoten also nicht erst zur Zeit einer Abfrage berechnet, sondern werden explizit abgespeichert. Das ermöglicht Relationen in konstanter Zeit zu traversieren. Für die Analyse von Graphstrukturen ist das ein erheblicher Vorteil. Das zeigt auch Ian Robinson anhand eines Beispiels von Abfragen verschiedener Tiefen [RWE15]. Abfragetiefe steht für die Anzahl der Relationen, die traversiert werden müssen, um eine gegebene Abfrage beantworten zu können. Die in Abbildung 2.10 dargestellte Abfrage nach den Freunden eines Freundes, ist somit eine Abfrage der Tiefe 2. In Tabelle 2.1 wird erkenntlich, dass Graphdatenbanken spätestens ab einer Abfragetiefe von 3 deutlich schneller Resultate liefern können als es mit einem RDBMS möglich ist.

Depth	RDBMS Execution Time(s)	Neo4j Execution Time(s)	Records Returned
2	0.016	0.01	$\simeq 2500$
3	30.267	0.168	$\simeq 110,000$
4	1543.505	1.359	$\simeq 600,000$
5	$Unfinished$	2.132	$\simeq 800,000$

Tabelle 2.1: RDBMS vs. Neo4j Abfrage Reaktionszeiten [RWE15]

Als Abfragesprache dient Neo4j die deklarative Property Graph Abfragesprache Cypher.

2.5.2 Cypher

Cypher ist eine deklarative Abfragesprache für Property Graphen [Fra+18]. Initial wurde Cypher von der Neo4j Inc. exklusiv für die hauseigene Graphdatenbank Neo4j entwickelt. Seit 2015 wird die Sprache von der openCypher Implementers Group unter dem Namen openCypher als Open Source Projekt weiterentwickelt. Die Spezifikation der Sprache ist unter der Apache 2.0 Lizenz veröffentlicht. Die erste Version unter Leitung der openCypher Implementers Group ist openCypher Version 9 [Imp18]. Zum Stand dieser Arbeit ist openCypher Version 9 die aktuellste Version der Abfragesprache. Im weiteren Verlauf dieser Arbeit werden die Namen Cypher und openCypher synonym verwendet.

Cypher enthält Keywords und Klauseln für alle gängigen CRUD (Create, Read, Update, Delete) Operationen. Diese wurden in Cypher auf das Property Graph Modell angepasst. Die für die Abfrage von Resultaten wichtigsten Keywords und Klauseln sind MATCH, WHERE sowie RETURN.

MATCH Mittels der MATCH Klausel einer Abfrage, können Muster eines Graphen beschrieben werden, nach denen in der Datenbasis gesucht werden soll. Muster sind beliebig lange Verkettungen verschiedener Knoten mittels bestimmter Kanten. Um eine möglichst intuitive textbasierte Darstellung von Graphmustern zu ermöglichen, können Muster in Cypher mittels ASCII-Art beschrieben werden.

WHERE Im WHERE Part einer Abfrage können Bedingungen festgelegt werden, nach denen die Resultate einer Abfrage gefiltert werden. Beispielsweise können so die Namen der Personen, deren Alter unter einer bestimmte Grenze liegt, abgefragt werden. Die Funktionalität der Cypher WHERE Klausel ist mit der SQL WHERE Klausel bzw. der Operation der Selektion der relationalen Algebra vergleichbar.

RETURN Im RETURN Part der Abfrage wird festgelegt, welche Properties, Knoten und Kanten als Ergebnismenge dem Client zurückgegeben werden. Die Funktion der Klausel ist mit der der Projektion der Relationalen Algebra vergleichbar.

Neben den vorgestellten Klauseln, stellt Cypher Klauseln zur Verfügung mit denen Knoten und Kanten kreiert oder gelöscht werden können, Properties hinzugefügt, verändert oder entfernt werden können und Aggregate oder andere Funktionen berechnet werden können.

In Quelltext Abbildung 2.1 wird exemplarisch dargestellt, wie der in Abbildung 2.12 verwendete Property Graph, mittels Cypher, erzeugt werden kann. Dazu wird die Klausel CREATE verwendet. Im ersten CREATE Statement werden die Knoten des Graphen mit Labeln und Attributen erstellt. Das Label eines Knoten wird mit einem Doppelpunkt vom Knotenbezeichner abgetrennt. Der Knotenbezeichner kann in der Query als Referenz auf den erstellten Knoten verwendet werden. Die Key-Value Paare der Knotenattribute werden in einer, von geschweiften Klammern umfassten, Auflistung kodiert. Im zweiten CREATE Statement werden die Relationen zwischen den Knoten angelegt. Dazu werden die im vorherigen Statement verwendeten Knotenbezeichner wiederverwendet und durch mit ASCII-Art dargestellte Relationen miteinander verbunden. Wie auch bei der Erzeugung der Knoten, ist der Relationstyp durch einen Doppelpunkt von einem eventuellen Relationsbezeichner abgetrennt. Dieser wird hier nicht verwendet, da die angelegten Relationen im weiteren Verlauf der Query nicht wieder verwendet werden.

```
1  // Create Nodes & Labels & Properties
2  CREATE
3      (alice:Person {name: "Alice", age: 30}),
4      (bob:Person {name: "Bob", age: 22}),
5      (carol:Person {name: "Carol", age: 42}),
6      (dan:Person {name: "Dan", age: 35}),
7      (erin:Person {name: "Erin", age: 26}),
8      (frank:Person {name: "Frank", age: 57})
9
10 // Create Edges
11 CREATE
12     (alice)-[:IS_FRIENDS_WITH]->(bob),
13     (alice)-[:IS_FRIENDS_WITH]->(carol),
14     (bob)-[:IS_FRIENDS_WITH]->(dan),
15     (carol)-[:IS_FRIENDS_WITH]->(erin),
16     (carol)-[:IS_FRIENDS_WITH]->(frank)
```

Quelltext 2.1: Erzeugen des in Abbildung 2.12 dargestellten Property Graphen durch Cypher Query

Quelltext Abbildung 2.2 greift die in Abbildung 2.10 vorgestellte Abfrage nach den Namen der Freunde von Freunden von Alice erneut auf und stellt diese in Cypher Syntax dar. Dazu werden zwei Varianten der gleichen Abfrage vorgestellt. Die erste Variante, verwendet ein naives Graphmuster im MATCH Part der Abfrage. Vom Knoten `alice` der im `name`-Attribut den Wert `"Alice"` trägt, werden alle ausgehenden Kanten des Kantentyps `IS_FRIENDS_WITH` traversiert. Ausgehend von den darüber erreichten Knoten werden erneut alle von diesen Knoten ausgehenden `IS_FRIENDS_WITH` Relationen traversiert, die schließlich zu den Freunden der Freunde des Ausgangsknotens führen. Die zweite Variante verwendet ein sogenanntes Path Pattern. Mittels Path Patterns können Kantenzüge beliebiger Länge, die aus verschiedenen Relationstypen bestehen, als Graphmuster angegeben werden. Die Frage nach den Freunden der Freunde von Alice lässt sich mit Path Patterns als Frage nach den Namen derjenigen Knoten, die ausgehend von Alice über einen Kantenzug der Länge zwei, bestehend aus `IS_FRIENDS_WITH` Relationen, erreichbar sind, reduzieren.

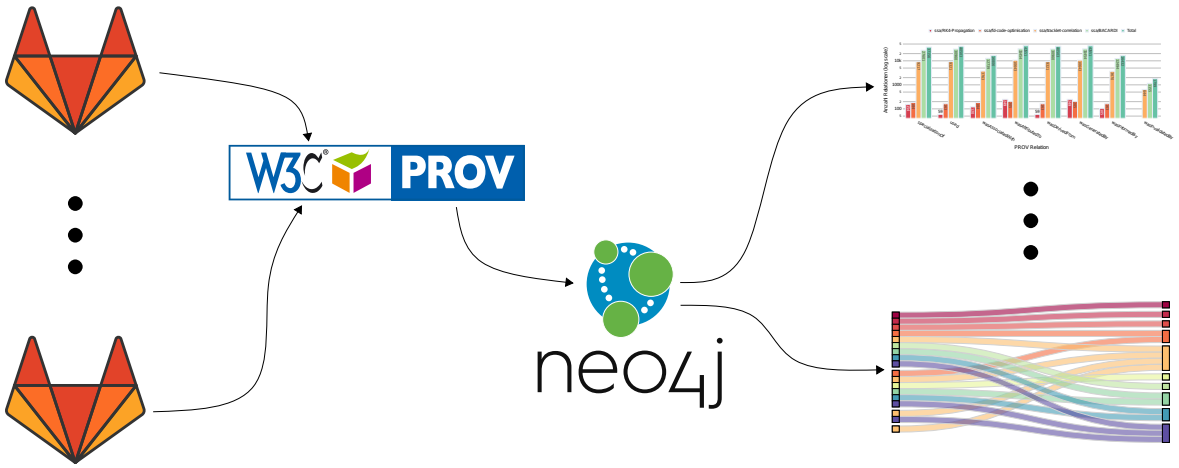
```
    // Friends of Friends of Alice
MATCH (alice)-[:IS_FRIENDS_WITH]->(friend),
      (friend)-[:IS_FRIENDS_WITH]->(fof)
WHERE  alice.name = "Alice"
RETURN alice.name AS aliceName, fof.name AS fofName

// Friends of Friends of Alice Using Path Pattern Lengths
MATCH (alice)-[:IS_FRIENDS_WITH*2]->(fof)
WHERE  alice.name = "Alice"
RETURN alice.name AS aliceName, fof.name AS fofName
```

Quelltext 2.2: Wer sind die Freunde der Freunde von Alice? Multi-Hop Cypher Query der Abb. 2.10 auf dem in Abb. 2.12 dargestellten Property Graphen

Damit endet das Grundlagenkapitel. Im folgenden Kapitel wird das in dieser Arbeit entwickelte Konzept zur provenienz-basierten Prozessanalyse von GitLab Projekten vorgestellt.

3 Methodik



Für diese Arbeit werden mit dem Python Tool GitLab2PROV Provenienzgraphen zu bestimmten GitLab Projekten generiert. Die Provenienzgraphen der Projekte werden anschließend vorverarbeitet und in die Graphdatenbank Neo4j importiert. Über das Cypher Interface der Graphdatenbank Neo4j werden Abfragen zur Graphstruktur, Entwicklung der Graphstruktur und zu Prozess spezifischen Fragestellungen ausgeführt. Die Ergebnisse der Abfragen werden mit der Visualisierungs Library Plotly graphisch aufbereitet und anschließend interpretiert. Für die Fragestellungen zur Graphstruktur und Entwicklung der Graphstruktur wird auf mehrere GitLab Projekte eingegangen, um die Skalierbarkeit des Analyseansatzes darzustellen. Die prozessspezifischen Fragestellungen werden anhand des GitLab Projektes der DLR Software BACARDI demonstriert, wobei die Abfrageergebnisse auf Auswirkungen untersucht werden, welche die Einführung des neuen Entwicklungsprozesses auf das BACARDI Projekt hat.

Der gewählte Ansatz besteht aus vier Teilschritten. Diese sind die Generierung des Provenienzgraphen, die Vorverarbeitung des Graphen, der Import des Graphen in die Graphdatenbank Neo4j und die Analyse des Graphen durch Cypher Queries. Die Analyse wird durch festgelegte Fragestellungen unterstützt. Im Folgenden werden die Teilschritte erläutert.

3.1 Provenienzgraphgenerierung

Zur Generierung der Provenienzgraphen im W3C PROV Standard wird das in Abschnitt 2.4 vorgestellte Python Tool GitLab2PROV verwendet. Unter Angabe eines API Tokens der GitLab Instanz, die die Projekte enthält deren Provenienzgraphen generiert werden sollen, fragt GitLab2PROV Metadaten und Artefakte der Projekte via HTTP Requests über die GitLab API ab.

Die API einer GitLab Instanz ist standardmäßig auf zehn Requests pro Sekunde pro IP limitiert. Wird die GitLab Instanz selbst gehostet, kann das Limit der Abfragen pro Sekunde aufgehoben werden. Um die Datenbeschaffung zu beschleunigen, kann das Rate Limit der API in der Konfiguration von GitLab2PROV angegeben werden. Anschließend führt GitLab2PROV die zur Datenbeschaffung nötigen HTTP Requests asynchron aus, ohne das angegebene Rate Limit zu überschreiten. Dazu wird die Python Library aiohttp [Kim19] verwendet.

Die Daten, die GitLab2PROV abfragt, werden gemäß der in Abschnitt 2.4 vorgestellten Modelle aufbereitet und in einem W3C PROV Serialisierungsformat abgespeichert. In dieser Arbeit wird das Serialisierungsformat PROV-JSON verwendet, in welchem Provenienzdaten des PROV Standardes im JSON Format serialisiert werden, um den in Schritt drei folgenden Import in die Graphdatenbank Neo4j zu vereinfachen.

Neben der Generierung einzelner Graphen, bietet GitLab2PROV die Möglichkeit den Provenienzgraph mehrerer GitLab Projekte zu genererieren. Dieser besteht aus den zusammengefügten Provenienzgraphen der einzelnen GitLab Projekte. Die interne Repräsentation des Provenienzgraphen sowie das Zusammenfügen mehrerer Graphen zu einem Graphen, wird durch das Python Package prov unterstützt [Huy18].

Der Provenienzgraph eines Projektes umfasst die am Projekt beteiligten Akteure, die von den Akteuren ausgeführten Aktionen und die durch Aktionen anfallenden Artefakte und Artefaktversionen sowie semantische Beziehungen zwischen den einzelnen Datenpunkten. Damit dokumentiert der Provenienzgraph eines GitLab Projektes den Entstehungsprozess des Projektes.

Nach der Generierung des Provenienzgraphen wird dieser weiter verarbeitet, um einen möglichst idealen Zustand zum Zeitpunkt der Analyse zu gewährleisten. Welche Schritte dabei unternommen werden, wird im nächsten Absatz geschildert.

3.2 Provenienzgraphvorverarbeitung

Nach der Generierung des Provenienzgraphen eines GitLab Projektes, wird der Graph vorverarbeitet, um einen möglichst idealen Zustand zum Zeitpunkt der Analyse zu gewährleisten. Dazu wird zunächst manuell ein Problem gelöst, das GitLab2PROV nicht automatisiert lösen kann.

GitLab zeichnet für Git Aktionen, wie dem Committen von Dateiänderungen den für die Aktion verantwortlichen Nutzer auf. Für Git Aktionen speichert GitLab den Namen und die E-Mail Adresse des verantwortlichen Nutzers, die der Nutzer in seiner lokalen Git Konfigurationsdatei festgelegt hat. Für Aktionen, die von Nutzern auf dem GitLab Webinterface getätigt werden, werden der Namen und die E-Mail Adresse gepseichert, die der verantwortliche Nutzer für seinen GitLab Account verwendet. Verwendet eine Person unterschiedliche Namen und E-Mail Adressen für die Git Konfiguration und den GitLab Account, können die Git Aktionen und GitLab Aktionen nicht automatisiert auf die gleiche Person zurückgeführt werden.

GitLab2PROV erzeugt in einer solchen Situation zwei unterschiedliche Agents. Einen, der für die Git Aktionen verantwortlich ist, sowie einen der für die GitLab Aktionen verantwortlich ist. Beide sollen die gleiche Person repräsentieren, können jedoch ohne weitere Informationen nicht zusammengeführt werden. Für Nutzer, die zwischenzeitlich ihren Namen ändern, können sogar mehr als zwei verschiedene Agents anfallen. In der Analyse ist dies durchaus von Nachteil, da allein durch die Anzahl der unterschiedlichen Agents nicht mehr auf die Anzahl der unterschiedlichen Personen geschlossen werden kann, die an einem Projekt beteiligt sind.

Um duplizierte Agents wieder zu vereinen, kann ein Alias Mapping angegeben werden. Das Alias Mapping ist eine Abbildung von Klarnamen von Nutzern auf die verschiedenen Aliasnamen, die die duplizierten Agents, welche den gleichen Nutzer repräsentieren, tragen. Nutzt z. B. Robert in seiner Git Konfiguration den Namen “Bob”, in seinem GitLab Account jedoch den Namen “Robert”, kann durch eine Abbildung des Namen “Robert” auf den Namen “Bob” dafür gesorgt werden, dass GitLab2PROV erkennt, welche Namen und damit welche Agents die gleiche Person repräsentieren. Diese können anschließend zusammengeführt werden.

Das Zusammenführen der Agents wird durch Graph Rewriting vorgenommen. Gegeben ein Alias Mapping und einen GitLab2PROV Provenienzgraphen, werden zunächst alle Agents des Graphen vereint, die dem Mapping zufolge die gleiche Person repräsentieren. Anschließend werden alle Kanten des Graphen, die an einem nach Wiedervereinigung der Agents nicht mehr existenten Agent starten oder enden, aktualisiert, so dass die Kante am neu vereinten Agent startet bzw. endet. Ein Beispiel des Graph Rewritings auf der Basis eines Alias Mappings, ist in Abbildung 3.1 dargestellt.

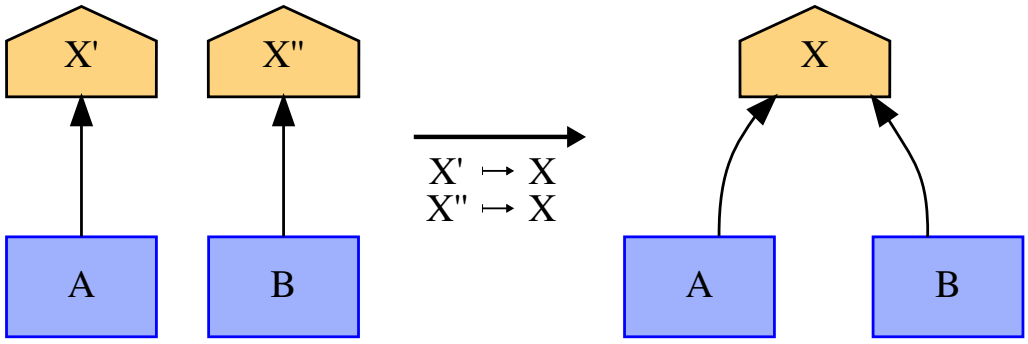


Abbildung 3.1: Graph Rewriting zur Zusammenführung von PROV Agents

Zusätzlich zum Zusammenführen der Duplikate von Agents, werden die Agents des Graphen pseudonymisiert. Die Pseudonymisierung ist kein verpflichtender Schritt in der Analyse der Projekte, wird in dieser Arbeit jedoch vorgenommen, da eventuelle Ergebnisse der Auswertung der Projekte, die einer Performance Evaluation einzelner Nutzer entsprechen, nicht negativ auf die Nutzer zurückfallen sollen. Zur Pseudonymisierung werden die Klarnamen des Alias Mappings durchnummeriert. Nach Zusammenführung der Agents tragen diese nur noch eine Zahl als Namen. Andere personenbezogene Informationen, wie z. B. E-Mail Adressen, werden bei der Zusammenführung der Agents entfernt und gelöscht.

Im nächsten Schritt wird der fertig verarbeitete Provenienzgraph in die Graphdatenbank Neo4j importiert. Welche Tools dabei zur Verfügung stehen und welche verwendet werden, wird im nächsten Absatz erklärt.

3.3 Neo4j Graph Import

Nach erfolgreicher Generierung und Vorverarbeitung wird der Provenienzgraph in die Graphdatenbank Neo4j importiert.

Provenienzgraphen des W3C PROV Standards beruhen auf RDF und damit auf dem RDF Graph Modell. Die Graphdatenbank Neo4j jedoch basiert auf dem Labelled Property Graphmodell (LPG). Zum Import vom Graphen des RDF Graph Modells muss eine Abbildung vom RDF Graphmodell auf das LPG Modell vorgenommen werden.

Zum Import und der damit verbundenen Abbildung existieren verschiedene Tools. Beispielsweise das im DLR entwickelte Python Tool prov-db-connector, welches als Verbindung zwischen Provenienzgraphen und verschiedenen Datenbanken wie Neo4j oder auch RedisGraph[1] dient [Bie+20]. prov-db-connector bildet den gesamten PROV Standard ab und findet als Document Store zum Speichern von Provenienzgraphen Verwendung. Die Geschwindigkeit des Graphimports sowie die Skalierbarkeit der Abbildung des RDF Modells auf das LPG Modell über die Größe der zu importierenden Graphen, wurde bei der Entwicklung von prov-db-connector nicht priorisiert.

Um das Testen von Graphstrukturen bei der Entwicklung von GitLab2PROV zu beschleunigen und einen skalierbareren Ansatz zum Import von Provenienzgraphen in die Graphdatenbank Neo4j zu implementieren, wird im DLR das Python Tool PROV2Neo entwickelt[2]. PROV2Neo nutzt zum Import von Provenienzgraphen den Object Graph Mapper (OGM) des Python Packages Py2Neo [Sma19], dessen Konzept dem eines Object Relational Mappers (ORM) einer relationalen Datenbank gleicht. Durch den OGM kann mit Graphkonzepten, wie Knoten und Kanten, auf Basis nativer Python Objekte gearbeitet werden, ohne komplexe Cypher Queries schreiben zu müssen. Zur Arbeit mit Provenienzdaten des W3C PROV Standards, nutzt PROV2Neo das Python Package prov [Huy18].

PROV2Neo befindet sich Anfangsstadium der Entwicklung und unterstützt zum Zeitpunkt dieser Arbeit noch nicht alle Features, die in der Spezifikation des W3C PROV Standards erwähnt werden. Aufgrund der Größe des in dieser Arbeit verwendeten Graphen, wird PROV2Neo dennoch bereits in diesem Stadium zum Import des Graphen in die Graphdatenbank Neo4j verwendet.

Als weitere Alternative zur Arbeit mit RDF Daten mit der Graphdatenbank Neo4j sei an dieser Stelle das Neo4j Plugin neosemantics erwähnt [BC20]. Mit neosemantics können beliebige Datensätze des RDF Formates in die Graphdatenbank Neo4j importiert werden.

Im nächsten Schritt, der Analyse des Provenienzgraphen, werden unterschiedliche Teilbereiche, wie z. B. die Struktur des Graphen, betrachtet. Die verschiedenen Teilbereiche der Analyse werden im folgenden Absatz näher erläutert.

3.4 Provenienzgraphanalyse

Zur Analyse des Provenienzgraphen werden drei verschiedene Teilbereiche näher betrachtet: Zuerst die Graphstruktur, anhand der Anzahl der Knoten und Kanten des Graphen. Anschließend wird die Entwicklung der Graphstruktur über den Zeitraum der Projekte, die im Provenienzgraphen enthalten sind, untersucht. Zuletzt werden einige prozessspezifische Fragestellungen, wie die Anzahl der Entwickler, die an Teilprojekten mitarbeiten, geklärt.

[1] `https://oss.redislabs.com/redisgraph/`
[2] `https://gitlab.dlr.de/provenance/prov2neo`

Die Fragestellungen der ersten beiden Teilbereiche werden anhand mehrerer GitLab Projekte ausgewertet. Damit soll gezeigt werden, dass die vorgestellte Analysemethodik auf beliebig viele Projekte in einem einheitlichen Datenmodell skalierbar ist. Die Auswertung der letzten Fragestellung wird auf das GitLab Projekt der DLR Software BACARDI beschränkt. Eine Auswertung für jedes der vorher verwendeten Projekte ist prinzipiell möglich, würde den Rahmen dieser Arbeit jedoch übersteigen. Zudem kann bei der Festlegung des Fokus auf das BACARDI GitLab Projekt auf die Entwicklungsgeschichte des Projektes eingegangen und erläutert werden, welche Indizien der Entwicklungsgeschichte sich in den aufgezeichneten Provenienzdaten finden lassen.

Die Fragestellungen werden in Form von Cypher Queries formuliert, deren Ergebnisse mit Hilfe der Python Library Plotly visualisiert werden.

3.4.1 Graphstruktur

Bei der Auswertung der Graphstruktur des Provenienzgraphen wird darauf eingegangen, welche Aussagen sich aus der Anzahl der Knoten und Kanten des Graphen in Bezug auf die verwendeten Provenienzmodelle treffen lassen.

Anzahl Knoten

```
1  MATCH (n)
2  WHERE EXISTS (n.project)
3  WITH
4      n.project AS project,
5      LABELS(n)[0] AS provType,
6  RETURN
7      project,
8      provType,
9      COUNT(provtype) AS provtypeCount
10 ORDER BY provType, project ASC
```

Quelltext 3.1: Cypher Query - Anzahl Activities und Entities pro Teilprojekt

Die Anzahl der Knoten der PROV Grundtypen der GitLab Projekte, die im Provenienzgraphen enthalten sind, wird in der Cypher Query 3.1 abgefragt. Dabei werden zunächst die Knoten des PROV Typen Agent ausgelassen. Ein Knoten ist Teil des Provenienzgraphen eines Projektes, wenn der Identifikator des Projektes in den Eigenschaften des Knoten als Attributwert vorliegt oder wenn eine direkte Verbindung zwischen dem Knoten und einem anderen Knoten besteht, der den Identifikator des Projektes in seinen Knoteneigenschaften enthält.

Der Identifikator eines Projektes wird nur in den Knoteneigenschaften von Entities und Activities gespeichert. Nicht jedoch in den Eigenschaften von Agents, da Agents in mehereren Projekten tätig sein können und damit mehreren Projekten angehören könnnen. Die Darstellung von mehrwertigen Attributen wird von PROV2Neo noch nicht unterstützt, daher wird es vermieden die Projektidentifikatoren für Agents abzuspeichern.

Anhand der Anzahl der Knoten der PROV Grundtypen eines GitLab2PROV Provenienzgraphen, können mehrere Aussagen über das Projekt, welches im Graph abgebildet wird, getätigt werden.

Die Anzahl der Agents des Provenienzgraphen eines Projektes gibt z. B. Auskunft über die Anzahl der Personen die im Projekt aktiv waren. Die Anzahl der Activities spiegelt die Anzahl der Aktionen wieder, die im Projekt getätigt wurden. Die Anzahl der Entities wiederum gleicht der Anzahl der Artefakte und Artefaktversionen, die im Verlauf des Projektes entstanden sind.

Durch das Verhältnis von Entities pro Activity können zudem Aussagen über das Projekt unter Bezug auf die GitLab2PROV Provenienzmodelle getätigt werden. Bestimmte Modelle erlauben nur ein festes Verhältnis der Anzahl von Entities pro Activity. Je nach dem Verhältnis, das letztlich im Graphen vorliegt, kann darauf geschlossen werden, welches Modell am häufigsten eingesetzt wurde. Beispielsweise erlaubt das Commit Modell beliebig viele Entities pro Activity dem Graph hinzuzufügen (vgl. Abbildung 2.5). Das Modell für GitLab Webressourcen erzeugt stets nur eine Entity pro Activity, nur bei der Kreierung der Ressource werden einmalig zwei Entities pro Activity erzeugt (vgl. Abbildung 2.9). Enthält ein Provenienzgraph mehr als eine Entity pro Activity, ist klar, dass im Verlauf des Projektes ein Commit stattgefunden hat, der mehr als eine Datei hinzugefügt bzw. verändert hat.

Anzahl Kanten

```
1  MATCH (n)-[relationship]->()
2  WHERE EXISTS(n.project)
3  WITH
4      n.project AS project,
5      TYPE(relationship) AS relationshipType
6  RETURN
7      project,
8      relationshipType,
9      COUNT(relationshipType) AS relationshipTypeCount
10 ORDER BY relationshipType, project ASC
```

Quelltext 3.2: Cypher Query - Anzahl Relationen pro Teilprojekt

Neben den Knoten eines Graphen, gehören die Kanten des Graphen zu den strukturgebenden Eigenschaften. In der in Quelltext Abbildung 3.2 dargestellten Cypher Query wird nach der Anzahl der Relationen der einzelnen Teilprojekte eines Provenienzgraphen gefragt. Anhand der Anzahl der Relationen bestimmter Typen können, in Bezug auf die Provenienzmodelle, ähnlich zum Verhältniss von Entities pro Activity, Aussagen über das GitLab Projekt getätigt werden, welches im Provenienzgraph abgebildet wird.

So kommt zum Beispiel die Relation *wasInvalidatedBy* nur im GitLab2PROV Modell des Entfernens einer Datei durch einen Commit vor. Ist keine Relation dieses Typen im Provenienzgraphen des Projektes enthalten, kann ausgesagt werden, dass im gesamten Projektzeitraum kein Commit stattfand, der eine Datei entfernte.

Ähnlich können Aussagen über das Interaktionsverhalten der am Projekt beteiligten Personen anhand der Anzahl der Relationen *used* und *wasGeneratedBy* getroffen werden. Die Relation *used* wird im Kontext der Erstellung neuer Versionen von Artefakten, wie Dateien oder Issues, unter Verwendung von vorherigen Versionen verwendet. Die Relation *wasGeneratedBy* wird im Kontext der Erstellung von neuen Artefakten verwendet. Ist die Anzahl der *used* Relationen um ein Vielfaches geringer als die Anzahl der *wasGeneratedBy* Relationen, kann ausgesagt werden, dass im betrachteten Projekt mehr Aktionen stattfinden, die kom-

plett neue Artefakte generieren als Aktionen die neue Versionen bereits erstellter Artefakte produzieren.

3.4.2 Graphstrukturentwicklung

Die Graphstruktur des Provenienzgraphen eines Projektes, die im vorherigen Abschnitt im Zentrum der Analyse stand, ist eine Momentaufnahme des Projektes zum Zeitpunkt der Generierung des Provenienzgraphen. Wie sich der Provenienzgraph eines Projektes über die Zeit entwickelt, gibt Aufschluss über die Aktivität des im Graphen abgebildeten GitLab Projektes.

Anzahl Knoten über Zeit

```
1  MATCH (activity:Activity)
2  RETURN
3      ID(activity) AS id,
4      LABELS(activity)[0] AS provType,
5      activity.project AS project,
6      activity.prov:startTime AS time
7  UNION ALL
8  MATCH
9      (entOrAgt)-[:wasGeneratedBy|wasAssociatedWith]-(activity:Activity)
10 RETURN
11     ID(entOrAgt) AS id,
12     LABELS(entOrAgt)[0] AS provType,
13     activity.project AS project,
14     activity.prov:startTime AS time
```

Quelltext 3.3: Cypher Query - Anzahl Knoten pro Teilprojekt über Projektzeitraum

Für Provenienzgraphen lässt sich Wachstum als Zuwachs der Anzahl der Knoten des Graphen über Zeit definieren. Dafür wird jedem Knoten ein Zeitpunkt zugewiesen, zu dem der Knoten der Knotenmenge des Graphen hinzugefügt wurde.

Für PROV Activities ist der Zeitpunkt, zu dem die Activity dem Graph hinzugefügt wird, klar definiert. Jede Activity im GitLab2PROV Modell trägt einen Startzeitpunkt, dieser kann als Zeitpunkt angesehen werden, zu dem die Activity der Knotenmenge des Graphen beitritt. Für Agents und Entities wird kein Startzeitpunkt festgehalten. Entities und Agents sind jedoch direkt mit Activities verbunden. Erstere werden von Activities generiert und letztere verantworten Activities. Für eine gegebene Entity kann der Zeitpunkt des Beitritts zur Knotenmenge als Startzeitpunkt der Activity definiert werden, von welcher die Entity generiert wird. Für einen Agent hingegen ist der Zeitpunkt des Beitritts zur Knotenmenge des Graphen der Startzeitpunkt der ersten bzw. frühesten Activity, die mit dem Agent assoziiert wird. Dieser Zeitpunkt entspricht dem Zeitpunkt zu dem der Agent das erste Mal eine Activity verantwortet und zum ersten Mal in dem Projekt aktiv wird, in welchem die Activity stattfindet.

In Abbildung 3.3 ist die Cypher Query dargestellt, die verwendet wird, um für jeden Knoten des Provenienzgraphen den Zeitpunkt festzustellen, zu dem der Knoten der Knotenmenge des Graphen beigetreten ist.

Zurückgegeben wird für jeden Knoten die Neo4j interne ID, das Label, welches dem PROV Typen des Knoten entspricht, das Projekt des Provenienzgraphen zu welchem der Knoten gehört, sowie die Zeit zu der der Knoten der Knotenmenge des Graphen beigetreten ist. Die Zeitpunkte des Beitritts werden dabei wie zuvor beschrieben definiert.

3.4.3 Prozessspezifische Fragestellungen

Abgesehen von der allgemeinen Graphstruktur und ihrer Entwicklung, können Aussagen über die im Provenienzgraphen dargestellten Projekte durch spezifischere Fragestellung mit Hilfe des Provenienzgraphen beantwortet werden.

Wer wirkt an welchen Projekten mit?

```
1 MATCH (agent:Agent)<-[:wasAssociatedWith]-(activity:Activity)
2 RETURN
3     agent.user_name AS name,
4     COLLECT(DISTINCT activity.project) AS projects
```

Quelltext 3.4: Cypher Query - Agents pro Project

Gerade bei der Analyse mehrerer Projekte ist es von Interesse herauszufinden, wer an welchen Projekten mitwirkt und welche Projekte sich gegebenenfalls Entwickler teilen. So kann, unter anderem, ermittelt werden, welche Projekte wie miteinander in Verbindung stehen und an was für Projekten einzelne Entwickler mitarbeiten. Für die Bewertung der Rolle, welche ein Entwickler im Rahmen eines Projektes einnimmt, ist diese Information von Interesse.

In Quelltext Abbildung 3.4 ist die Cypher Abfrage dargestellt, mit welcher zu jedem Agent des Provenienzgraphen die Liste an Projektidentifikatoren der Projekte abgefragt wird, an denen der Agent beteiligt ist. In den Knoteneigenschaften der Knoten des Graphen, die Agents repräsentieren, werden keine Projektidentifikatoren gespeichert. Da Agents jedoch Activities verantworten, die wiederum Projektindentifikatoren tragen, können für die Agents die Projektidentifikatoren der Activities verwendet werden.

Wer ist wann in einem Projekt aktiv?

```
1  MATCH (agent:Agent)<-[:wasAssociatedWith]-(activity:Activity)
2  WHERE
3      EXISTS(activity.project) AND
4      EXISTS(activity.prov:startTime)
5  WITH
6      agent.user_name AS name,
7      activity.project AS project,
8      datetime(activity.prov:startTime) AS time,
9  RETURN
10     name,
11     project,
12     MIN(time) AS timeOfFirstActivity,
13     MAX(time) AS timeOfLastActivity
```

Quelltext 3.5: Cypher Query - Zeitpunkte der ersten und letzten Activity pro Agent pro Projekt

Die Anzahl der Personen, die zeitgleich an einem Projekt arbeiten, kann über den Verlauf eines Projektes stark variieren. Je nach Ambition des Projektes, Umfang der Geldmittel und Nähe der Deadlines verändert sich typischerweise die Anzahl der Personen, die zeitgleich an einem Projekt arbeiten. Über den zeitlichen Verlauf eines Projektes können viele verschiedene Personen am Projekt beteiligt sein. Um herauszufinden, wann wie viele Personen zeitgleich an einem Projekt arbeiten, kann die Anzahl der am Projekt beteiligten Personen zeitlich aufgelöst werden.

Zur zeitlichen Auflösung wird ein Aktivitätskriterium definiert, welches festlegt, in welchem Zeitraum eine Person aktiv an einem Projekt beteiligt ist. Dafür wird in dieser Arbeit der Zeitraum zwischen der ersten und letzten Aktion, die ein Nutzer in einem Projekt tätigt, verwendet. Auf den Provenienzgraphen des Projektes übertragen, entspricht dies dem Zeitraum zwischen der ersten und letzten Activity, die ein Agent in einem Projekt verantwortet.

Die in Quelltext Abbildung 3.5 dargestellte Cypher Query gibt für jeden Agent aller Projekte den Zeitpunkt der ersten bzw. frühesten und letzten bzw. spätesten Activity aus, mit welcher der jeweilige Agent assoziiert wird. Ausgehend von den Ergebnissen der Abfrage, können die Aktivitätszeiträume der Agents eines Projektes visualisiert werden. Zur Visualisierung wird ein Gantt Diagramm verwendet, welches im Evaluationsteil der Arbeit näher beschrieben wird.

Zeitleisten der Nutzerinteraktionen

```
MATCH (agent:Agent)<-[:wasAssociatedWith]-(activity:Activity)
WHERE activity.prov:type = "event"
RETURN
    activity.project AS project,
    activity.event_event AS eventType,
    datetime(activity.prov:startTime) AS time,
    agent.user_name AS name
UNION ALL
MATCH (agent:Agent)<-[:wasAssociatedWith]-(activity:Activity)
WHERE activity.prov:type IN [
        "commit",
        "issue_creation",
        "merge_request_creation"]
RETURN
    activity.project AS project,
    activity.prov:type AS eventType,
    datetime(activity.prov:startTime) AS time,
    agent.user_name AS person
```

Quelltext 3.6: Cypher Query - Nutzerinteraktionen mit Zeit und verantwortlicher Person

Nutzern stehen vielfältige Interaktionsmöglichkeiten mit GitLab Projekten zur Verfügung. So kann ein GitLab Projekt nicht nur als Code Repository genutzt werden, sondern Nutzer können auch, gleich einem Ticket System, Issues anlegen und in Merge Requests über Code Änderungen diskutieren, bevor diese ihren Weg in die Code Base des Projektes finden. Darüber hinaus können unter anderem Issues gelabelt werden, um sie zu kategorisieren, Ressourcen, durch Erwähnungen in Kommentaren, miteinander verknüpft werden und Aufgaben an Nutzer verteilt werden, indem diese Issues zugewiesen bekommen.

Ein Überblick über das Nutzungsverhalten der Interaktionsmöglichkeiten der am Projekt beteiligten Personen gibt Einblick in gleich mehrere Punkte, die wiederrum Aussagen über den Entwicklungsprozess des Projektes zulassen.

1. Die Aktivität des Projektes anhand der Anzahl der Nutzerinteraktionen
2. Das Featureset, welches von beteiligten Personen genutzt wird
3. Die Veränderungen im Nutzungsverhalten bzw. Interaktionsverhalten
4. Die Rollen einzelner Nutzer anhand ihres Interaktionsverhaltens

Ein Teil der Interaktionen, die Nutzern zur Verfügung stehen, werden bei Gebrauch von GitLab dokumentiert. Die Dokumentation der Interaktionsevents nimmt GitLab in Form von sogenannten System Notes vor. Abbildung 3.2 zeigt beispielhaft, wie GitLab System Notes auf der Timeline eines Issues darstellt. Eine System Note beschreibt ein Interaktionsevent und zeichnet den Nutzer auf, der für dieses Event verantwortlich ist.

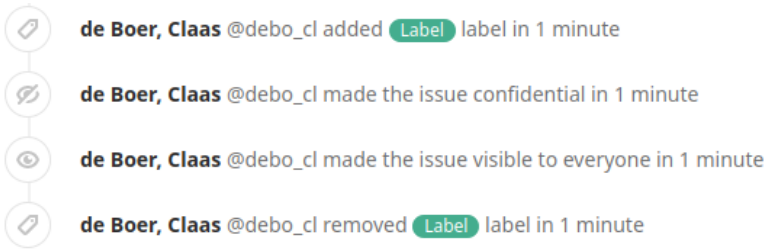


Abbildung 3.2: System Notes dokumentieren Nutzerinteraktionen

GitLab2PROV extrahiert die System Notes eines Projektes über die GitLab API, um Interaktionen von Entwicklern zu rekonstruieren und im Provenienzgraph aufzuzeichnen. Das Parsen der System Notes ist aufwändig, liefert jedoch mehr und umfassendere Informationen zu Events, die von Nutzern getätigt werden, als es die dedizierte GitLab Event API vermag. System Notes enthalten eine Event Zeichenkette, die mittels regulärer Ausdrücke geparsed werden kann. Eine Auflistung der Nutzerinteraktionen, welche GitLab2PROV aus System Notes parsed, findet sich im GitLab2PROV GitHub Projekt[3].

Die Zeichenkette eines Eventtypen kann sich über verschiedene GitLab Versionen hinweg verändern. Wird die Zeichenkette eines Eventtypen mit einem Versions Update aktualisiert, wird ab dem Zeitpunkt der Aktualisierung die neue Zeichenkette verwendet. Für alte Events des gleichen Typen wird die Zeichenkette nach Aktualisierung nicht nachträglich angepasst, sodass für ein und den selben Eventtypen mehrere Notationen parallel in Form von System Notes existieren können. Auf Beispiele, die diesen Umstand demonstrieren, wird im Evaluationsteil der Arbeit eingegangen.

Nutzerinteraktionen, wie das Committen von Dateiänderungen oder das Öffnen von Merge Requests und Issues, werden nicht in System Notes festgehalten, sie sind im GitLab2PROV Provenienzgraphen eines Projektes dennoch enthalten (vlg. Abschnitt 2.4).

Die in Quelltext Abbildung 3.6 dargestellte Cypher Query gibt den Zeitpunkt aller aufgezeichneten Nutzerinteraktion, sowie den Namen der für die Interaktion verantwortlichen Person zurück. Die Query umfasst Git Commits, das Öffnen von Issues, das Öffnen von Merge Requests sowie jegliche Art von Interaktionen, die durch System Notes aufgezeichnet werden. Für jeden Eventtypen kann, ausgehend vom Ergebnis der Query, eine Zeitleiste erstellt werden, auf der die Zeitpunkte eingezeichnet werden, zu denen Events bzw. Interaktionen der jeweiligen Typen stattfinden. Zusätzlich können die Events in verschiedenen Farben eingezeichnet werden, sodass die Farbe eines Events den Nutzer repräsentiert, der für das Event verantwortlich ist.

Im nächsten Teil der Arbeit, der Evaluation, wird die vorgestellte Methodik anhand von vier GitLab Projekten demonstriert. Bei der Analyse der Graphstruktur werden alle vier GitLab Projekte betrachet. Bei den projektspezifischen Fragestellungen wird sich auf das GitLab Projekt der DLR Software BACARDI beschränkt. Dabei wird darauf eingegangen, wie die Ergebnisse der Analyse die Einführung eines, anhand der DLR SE Guidelines entworfenen, Entwicklungsprozesses im BACARDI Projekt dokumentieren.

[3] `https://github.com/cdboer/ba-thesis-jupyter-notebook/releases/tag/v1.0`

4 Evaluation

Die im vorherigen Kapitel vorgestellte Methodik wird in diesem Kapitel an vier verschiedenen GitLab Projekten demonstriert. Dabei wird auf den Entwicklungsprozess der DLR Software BACARDI besonderes Augenmerk gelegt. Die Analyse der Graphstruktur wird für alle vier Projekte vorgenommen, die projektspezifischen Fragestellungen werden zum GitLab Projekt der DLR Software BACARDI ausgewertet.

Vor Beginn der Evaluation wird die DLR Software BACARDI vorgestellt und erläutert, warum das BACARDI GitLab Projekt ausgewählt wurde. Dazu wird der im BACARDI Projekt eingeführte Entwicklungsprozess mit den im Prozess enthaltenen Vorgaben zur Verwendung von GitLab beschrieben.

4.1 BACARDI

Der "Backbone Catalogue of Relational Debris Information" (BACARDI) ist eine vom DLR entwickelte Software, die unter anderem zur Bestimmung der Bahnen orbitaler Flugobjekte und zur Errechnung von Kollisionswarnungen aus den zuvor bestimmten Bahninformationen genutzt wird [Sto+19]. Entwickelt wird BACARDI in Kooperation des DLR Institutes für Softwaretechnologie (SC) sowie des DLR Institutes für Raumflugbetrieb und Astronautentraining (RB).

Beispiele orbitaler Flugobjekte sind Satelliten, die Internationale Raumstation ISS oder auch Weltraumschrott. Durch hohe relative Orbitgeschwindigkeiten von bis zu 15 km/s können beim Zusammenprall zweier Objekte hohe Energiemengen freigesetzt werden. Selbst beim Zusammenstoß mit Kleinstteilen können daher Flugobjekte wie Satelliten beschädigt oder zerstört werden. Für den Betrieb von Raumfahrtmissionen ist es daher wichtig, nicht nur über die eigenen Flugobjekte Bescheid zu wissen, sondern auch die Bahnen anderer Objekte zu kennen. Wird erkannt, dass sich zwei Objekte gefährlich nahe kommen, können Ausweichmanöver eingeleitet werden. BACARDI bildet, als hochaktueller Bahnkatalog, die Grundlage zur Berechnung von Kollisionswarnungen, die von Software Systemen, wie dem Collision Avoidance System (COLA) des German Space Operation Center (GSOC), weiterverarbeitet werden können.

Seit 2015 wird das BACARDI Projekt in einem GitLab Projekt einer DLR internen GitLab Instanz entwickelt. Nach Veröffentlichung der SE Guidelines des DLR [SMH18], wurde im Sommer 2018 anhand der Guidelines ein Entwicklungsprozess für das BACARDI Projekt entwor-

fen, der im Juli 2018 eingeführt wurde. Daher wurde das GitLab Projekt der Software BACARDI als Beispiel ausgewählt, um die Auswirkungen der Einführung eines neuen Entwicklungsprozesses zu untersuchen.

Der eingeführte Entwicklungsprozess gibt unter anderem vor, wie im Rahmen des BACARDI Projektes mit GitLab gearbeitet werden soll. Die Teile des Entwicklungsprozesses, welche die Interaktionen von Entwicklern mit GitLab betreffen, werden im Folgenden vorgestellt. Dazu werden zunächst einige Begriffe erläutert, die bei der Vorstellung des Entwicklungsprozesses verwendet werden.

- **Ticket Workflow:** Der Ticket Workflow beschreibt, wie im BACARDI Projekt mit Issues und offenen Aufgaben umgegangen wird, und wo die Issues und Aufgabenbeschreibungen abgelegt werden. Im BACARDI Projekt wird der Ticket Workflow mit Hilfe des GitLab Issue Tracker umgesetzt.
- **Definition of Done (DoD):** Die "Definition of Done" ist eine Checkliste, die bei der Entwicklung neuer Features und dem Review entwickelter Features verwendet werden kann. Die erste Hälfte der DoD unterstützt den Entwickler eines neuen Features. Die zweite Hälfte betrifft das Review und die Qualitätskontrolle des entwickelten Features und unterstützt den Reviewer des Features.
- **Git Workflow:** Der Git Workflow legt fest, wie im Rahmen des BACARDI Projektes mit dem Versionskontrollsystem Git umgegangen werden soll. Im Zentrum eines GitLab Projektes steht ein Git Repository, welches vom GitLab Projekt gewrapped und um neue Features erweitert wird. Im Git Workflow ist festgelegt, nach welchem Schema neue Feature Branches benannt werden und wie mit Feature Branches, sowie dem Baseline Branch verfahren wird.

Sprint Planung

Die Entwicklung der Software BACARDI wird in Sprints aufgeteilt. Ein Sprint ist eine kurze Zeitperiode, in der sich auf die Entwicklung vorher festgelegter Features konzentriert wird. Zur Aufzeichnung der Features, die in Zukunft implementiert werden sollen, wird der Issue Tracker des Projektes als Backlog verwendet. Wird ein neuer Sprint geplant, wird in mehreren Runden entschieden, welche Features in den Sprint aufgenommen werden. Dazu vergibt jeder der am Projekt beteiligten Entwickler den Issues im Backlog ein Gewicht von eins bis acht, welches die Entwickler in Form eines Kommentares festhalten. Eine Eins steht für niedrige Implementierungskomplexität und damit wenig Aufwand um den Issue umzusetzen. Eine Acht hingegen steht für hohe Komplexität und damit verbundenen hohen Zeitaufwand. Aus den angegebenen Gewichten eines Issues wird der Durchschnitt gebildet, welcher anschließend als Gewicht des Issues verwendet wird.

Für jeden Sprint steht ein Budget in Form einer Ganzzahl zur Verfügung, welches die schaffbare Arbeitslast eines Sprints approximiert darstellt. Ausgehend von der Größe des Budgets werden für einen Sprint Issues ausgesucht, sodass die Gesamtsumme der Gewichte der Issues die Größe des Budget nicht über- oder unterschreitet. Den ausgewählten Issues eines Sprints wird ein Milestone verliehen, der den Sprint repräsentiert, in welchem der Issue umgesetzt werden sollen.

Feature Entwicklung

Zu Beginn eines Sprints wählen die Entwickler des Projektes frei aus den für den Sprint bestimmten Features. Aus dem Issue des Features erstellt der für das Feature verantwortliche

Entwickler einen neuen Feature Branch sowie eine GitLab Merge Request. Anschließend beginnt die Entwicklung des Features auf dem neu erstellten Branch. Das Feature wird in inkrementellen Schritten implementiert und getestet. Dabei werden dem Feature Branch sowie der erstellten Merge Request Commits mit Dateiänderungen hinzugefügt. Der Entwickler des Features überprüft anhand der DoD, ob alle Aufgaben, die zur Implementierung des Features gehören, vollständig abgeschlossen sind. Ist das der Fall, beginnt das Review des frisch implementierten Feature.

Vor Beginn des Reviews, wird die Merge Request dem zuständigen Reviewer zugewiesen. Der verantwortliche Reviewer überprüft die Implementierung anhand der zweiten Hälfte der DoD-Checkliste. Zur Überprüfung gehört die Feststellung, ob der geschriebene Code den Code Guidelines entspricht, ob die geschriebenen Tests ausreichend sind, ob notwendige Dokumentation vorhanden ist, und ob der Code korrekt ist. Kommentare, die im Review entstehen, werden in der Merge Request festgehalten. Werden im Review Fehler gefunden, müssen diese behoben werden, bevor der Feature Branch in den Baseline Branch des Projektes gemerged werden kann. Sind alle Problemstellen behoben, markiert der Reviewer entstandene Diskussionen als abgeklärt, akzeptiert die Merge Request und führt den Merge des Feature Branches in den Baseline Branch aus. Die Merge Request wird anschließend dem für die Implementierung des Features verantwortlichen Entwickler zugewiesen.

Neben dem GitLab Projekt der BACARDI Software, werden zur Evaluation der in dieser Arbeit vorgestellten Methodik zur Prozessanalyse von GitLab Projekten drei weitere GitLab Projekte verwendet. Der Fokus der Analyse liegt jedoch auf dem BACARDI GitLab Projekt. Welche Projekte in die Evaluation einbezogen werden und auf welche Teile der Projekte Fokus gelegt wird, wird im nächsten Abschnitt erläutert.

4.2 Ergebnisse

GitLab Projekte der Evaluation

Zur Auswertung und Evaluation werden neben dem BACARDI GitLab Projekt drei weitere GitLab Projekte verwendet, um die Skalierbarkeit des vorgestellten Analyseansatzes zu demonstrieren. Die Namen der GitLab Projekte werden im Folgenden mit dem Präfix “ssa/” versehen, um zwischen dem GitLab Projekt ssa/BACARDI und dem DLR Projekt BACARDI unterscheiden zu können. Zur Evaluation werden die folgenden GitLab Projekte verwendet:

- ssa/BACARDI: Das GitLab Projekt ssa/BACARDI wird zur Entwicklung der zuvor beschriebenen BACARDI Software genutzt.
- ssa/tracklet-correlation: Das GitLab Projekt ssa/tracklet-correlation wird zur Entwicklung von Software verwendet, die Tracklets - Spuren von orbitalen Flugobjekten auf Langzeitbelichtungen aus passiv-optischen Beobachtungsverfahren - miteinander korrelieren, um festzustellen, ob sie zum gleichen Flugobjekt gehören. Das Projekt entstand im Rahmen einer Doktorarbeit und wird von Mitarbeitern der DLR HPC Gruppe weiterentwickelt.
- ssa/fd-code-optimisation: Das GitLab Projekt ssa/fd-code-optimisation wird zur Besprechung der Optimierung der von der BACARDI Software verwendeten Fortran Flight Dynamic Libraries genutzt. Das Projekt enthält keinerlei Source Code, da der Source Code der Flight Dynamic Libraries unter Geheimhaltung steht.
- ssa/RK4-Propagation: Das GitLab Projekt ssa/RK4-Propagation dient als Benchmark Projekt zum Test verschiedener Methoden zur Optimierung von Fortran-Codes durch

Parallelisierung auf Multiprozessorsystemen und der Aufteilung der Rechenlast der Codes auf CPU's und GPU's in Form eines hybriden Ansatzes.

Zur Evaluation wird zu jedem der genannten GitLab Projekte der Provenienzgraph des Projektes mit dem in Abschnitt 2.4 beschriebenen Tool GitLab2PROV generiert. Anschließend werden die Graphen der Projekte zu einem einheitlichen Graphen zusammengeführt. Agent Duplikate werden, wie im Methodikteil beschrieben, vereint oder ersetzt. Bei der Zusammenführung der Duplikate werden die Agents des Graphen pseudonymisiert. Zuletzt wird der Provenienzgraph durch PROV2Neo in eine Neo4j Instanz importiert. Der Versuchsablauf, die Cypher Queries und die Visualisierungen der Ergebnisse wurden in einem Jupyter Notebook dokumentiert, welches auf GitHub[1] sowie Zenodo zur Verfügung steht [Boe20]. Die Datensätze wurden am 2. Juli 2020 generiert. Alle Aktionen und Veränderungen an den Projekten, die nach dem 2. Juli 2020 stattgefunden haben, sind nicht in der Auswertung und in den generierten Provenienzgraphen enthalten.

4.2.1 Graphstruktur

Anzahl Knoten

Der generierte Provenienzgraph besteht aus den zusammengesetzten Provenienzgraphen der vorgestellten vier GitLab Projekte. Der Graph enthält insgesamt 61.446 Knoten sowie 238.301 Kanten. Wie sich die Knoten und Kanten auf die verschiedenen PROV Grundtypen und Provenienzgraphen der GitLab Projekte verteilen, stellt Abbildung 4.1 in Form eines Balkendiagrammes dar. Aufgrund der ungleichen Verteilung der Knoten auf die verschiedenen Projekte, ist die Y-Achse des Diagrammes logarithmisch skaliert. Bei der Auswertung werden PROV Agents vorerst ausgelassen, da Agents an mehreren Projekten beteiligt sein können und sich dieser Umstand nicht akkurat mit Hilfe eines Balkendiagrammes darstellen lässt. Die Betrachtung der Agents pro Projekt wird in einer der folgenden Fragestellungen nachgeholt. Damit wird eine Doppelung vermieden.

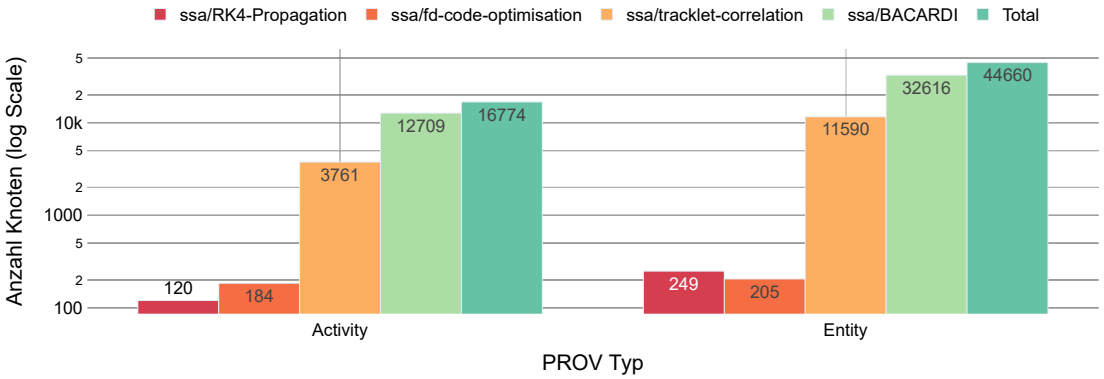


Abbildung 4.1: Anzahl der PROV Typen Activity und Entity der BACARDI Teilprojekte.

Der Provenienzgraph des GitLab Projektes ssa/BACARDI umfasst 12.709 Activities und 32.616 Entities. Mit 3.761 Activities und 11.590 Entities enthält der Provenienzgraph des GitLab Projektes ssa/tracklet-correlation nach dem des ssa/BACARDI Projektes die meisten Knoten der PROV Typen Activity und Entity. Darauf folgen das GitLab Projekt ssa/fd-code-optimisation

[1] https://github.com/cdboer/ba-thesis-jupyter-notebook/releases/tag/v1.0

mit 184 Activities und 205 Entities, sowie ssa/RK4-Propagation mit insgesamt 120 Activities und 249 Entities.

Daraus lässt sich schließen, dass im GitLab Projekt ssa/BACARDI die meisten Nutzerinteraktionen getätigt wurden, die im Provenienzmodell eines GitLab2PROV Provenienzgraphen als Activities dargestellt werden, sowie die meisten Artefakte und Artefaktversionen erstellt werden, welche durch Entitäten im Provenienzgraphen repräsentiert werden. Ob dabei die Anzahl der Artefakte oder die Anzahl der Artefaktversionen überwiegt, kann anhand der Anzahl der Knoten der PROV Grundtypen nicht ausgesagt werden.

Anhand des Verhältnisses der Entities pro Activity kann ausgesagt werden, welche der GitLab2PROV Modelle bei der Erstellung des Graphen zum Einsatz kamen. Da jedes der Modelle eine bestimmte Art der Interaktionen von Nutzern mit dem GitLab Projekt abbildet, kann ausgehend von den Eigenschaften der Modelle allein aufgrund der Graphstruktur bei einigen Modellen darauf geschlossen werden, welche Interaktionen Nutzer im Projekt getätigt haben müssen.

GitLab Projekt	Entities pro Activity
ssa/BACARDI	$\approx 2,566$
ssa/tracklet-correlation	$\approx 3,082$
ssa/fd-code-optimisation	$\approx 1,114$
ssa/RK4-Propagation	$2,075$

Tabelle 4.1: Anzahl der Entities pro Activity der GitLab2PROV Provenienzgraphen

Tabelle 4.1 stellt die Anzahl der Entities pro Activity der einzelnen Projektprovenienzgraphen dar. In drei der Projektprovenienzgraphen liegt die Anzahl der Entities pro Activity bei zwei oder höher. Die einzigen GitLab2PROV Modelle, die mehr als zwei Entities pro Activity erlauben, sind die Commit Modelle für das Hinzufügen neuer Dateien und das Hinzufügen von Dateiänderungen. Werden mehr als zwei Änderungen oder neue Dateien in einem einzelnen Commit hinzugefügt, generiert die Activity, die den Commit im Provenienzgraphen generiert, auch mehr als zwei Entities (vgl. Abbildung 2.5 und Abschnitt 2.4). Für die drei Projekte ssa/BACARDI, ssa/tracklet-correlation sowie ssa/RK4-Propagation kann dementsprechend allein aufgrund der Anzahl der Knoten ohne weitere Informationen ausgesagt werden, dass in den Projekten Commits getätigt wurden, die mehr als zwei Dateien hinzugefügt oder verändert haben.

Für das Projekt ssa/fd-code-optimisation kann keine solche Aussage mit Sicherheit getätigt werden. Auch die Umkehrung der Aussage, dass keine Commits getätigt wurden die Dateien hinzufügen oder verändern, ist nicht zwingend wahr. Immerhin könnte jeder Commit nur eine einzelne Änderung hinzufügen. Auch das würde eine Anzahl von weniger als zwei Entities pro Activity ermöglichen.

Anzahl Kanten

Neben den Knoten des Graphen, gehören die Kanten des Graphen zu den strukturgebenden Eigenschaften. Ähnlich der Anzahl der Knoten der PROV Grundtypen, gibt auch die Anzahl der Kanten der verschiedenen PROV Relationen Auskunft über die bei der Erstellung des Graphen verwendeten GitLab2PROV Modelle und damit über die abgebildeten Projektaktionen.

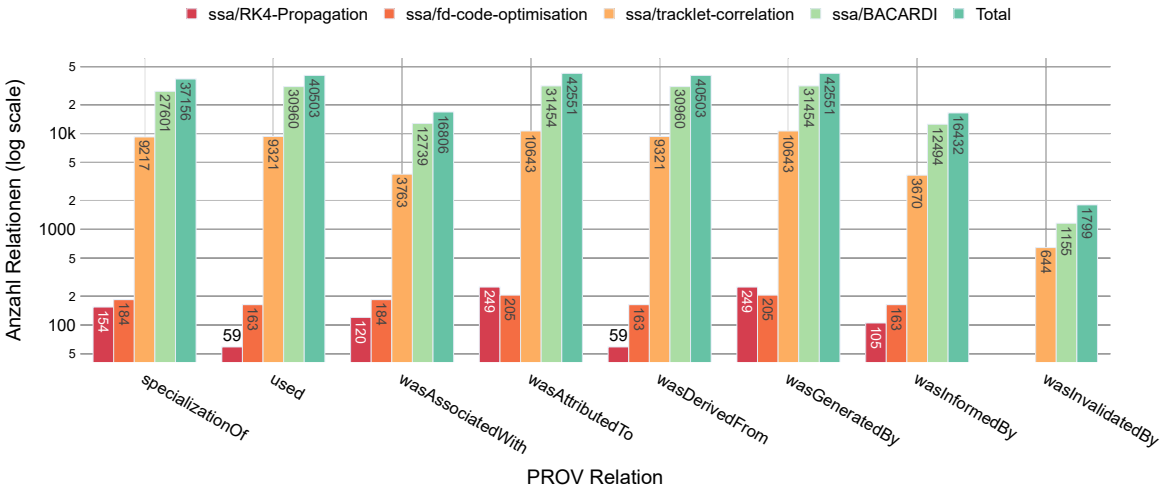


Abbildung 4.2: Anzahl der PROV Relationen der BACARDI Teilprojekte.

In Abbildung 4.2 wird die Anzahl der Kanten der PROV Relationen der einzelnen Projektprovenienzgraphen in einem Balkendiagramm mit logarithmisch skalierter Y-Achse dargestellt. Ähnlich zu den Knotentypen enthält der Provenienzgraph des GitLab Projektes ssa/BACARDI die meisten der aufgezeichneten PROV Relationen. Darauf folgt, wie auch in der Anzahl der Knoten, der Provenienzgraph des Projektes ssa/tracklet-correlation, der Provenienzgraph des Projektes ssa/fd-code-optimization sowie schließlich der Provenienzgraph des Projektes ssa/RK4-Propagation.

Im Vergleich der Graphen der Projekte ssa/fd-code-optimization sowie ssa/RK4-Propagation lassen sich verschiedene Beobachtungen machen. Beide Graphen enthalten keine *wasInvalidatedBy* Relationen. Relationen dieses Typen werden nur im GitLab2PROV Commit Modell des Entfernens einer Datei durch einen Commit verwendet. Daraus lässt sich schließen, dass in beiden Projekten keine Datei durch einen Commit entfernt wurde. Abgesehen davon enthält der Provenienzgraph des RK4-Propagation Projektes mehr Kanten einiger PROV Relationen, wie z. B. *wasGeneratedBy*, als der Projektprovenienzgraph des ssa/fd-code-optimisation Projektes, obwohl der ssa/fd-code-optimisation Graph insgesamt mehr Kanten enthält. Andere Relationstypen, wie z. B. *used*, sind hingegen im ssa/RK4-Propagation Projekt nicht so häufig vertreten wie im Graph des ssa/fd-code-optimization Projektes.

Die Relation *used* steht in Verbindung mit der Wiederverwendung von Artefakten bei der Generierung neuer Artefaktversionen, dargestellt durch Entities des Provenienzgraphen. Die Relation *wasGeneratedBy* hingegen wird sowohl bei der Kreierung neuer Artefakte als auch der Kreierung neuer Artefaktversionen verwendet. Im ssa/RK4-Propagation Projekt wurden demnach mehr Aktionen durchgeführt, die komplett neue Artefakte generieren als im ssa/fd-code-optimisation Projekt. Im Projekt ssa/fd-code-optimisation wurden dafür mehr Aktionen durchgeführt, die bereits bestehende Artefakte verwenden und neue Versionen dieser Artefakte erstellen.

Andere Relationen hingegen stehen in direkter Korrelation mit der Anzahl der Entities und Anzahl der Activities. So wird z. B. jede Entity von einer Activity generiert, für jede Entity existiert also eine *wasGeneratedBy* Relation. Von dieser Regel ausgenommen sind nur die Entities, die gelöschte Dateien repräsentieren (vgl. Abbildung 2.7). Ähnlich dazu wird jede Activity mit einem Agent über die Relation *wasAssociatedWith* assoziiert. Daher ist in allen Projektprovenienzgraphen die Anzahl der Activities gleich der Anzahl der *wasAssociatedWith* Relationen.

4.2.2 Graphstrukturentwicklung

Die Gesamtanzahl der Knoten und Kanten der Graphen ist eine Momentaufnahme des Provenienzgraphen der GitLab Projekte zum Zeitpunkt der Generierung der Graphen. Die zeitliche Auflösung des Wachstums des Graphen gibt Auskunft über die Projektaktivität. Das Wachstum eines Graphen wird in dieser Arbeit als Anstieg der Anzahl der Knoten eines Graphen über die Zeit definiert.

Anzahl Knoten über Zeit

Die Knoten eines Provenienzgraphen können Zeitinformationen tragen. So wird z. B. für Knoten des PROV Typen Activity der Startzeitpunkt der Activity in den Properties des Knoten abgespeichert. Dieser lässt sich unter dem Namen “prov:startTime” finden. Der Startzeitpunkt einer Activity wird im Rahmen dieser Arbeit als der Zeitpunkt interpretiert, zu dem die Activity der Knotenmenge des Graphen beigetreten ist. Vor Beginn der Activity existierte diese nicht im Graphen, erst nach Beginn der Activity gehört sie zur Knotenmenge. Damit lässt sich für Activities bestimmen, wann Activties der Knotenmenge eines Provenienzgraphen beitreten.

Für andere PROV Typen, wie Entities oder Agents, ist dies in den GitLab2PROV Provenienzmodellen nicht der Fall. Allerdings stehen Entities und Agents in direkter Relation zu Activities. So wird z. B. jede Activity mit einem Agent assoziiert (*wasAssociatedWith*). Gleichfalls wird jede Entity eines Provenienzgraphen von einer Activity generiert (*wasGeneratedBy*). Für Agents kann der Zeitpunkt, zu welchem der Agent der Knotenmenge des Graphen beitritt, dementsprechend als Startzeitpunkt der frühesten Activity definiert werden, welche mit dem Agent assoziiert wird. Für jede Entity kann der Zeitpunkt des Beitritts zur Knotenmenge des Graphen als Startzeitpunkt der Activity definiert werden, von welcher die Entity generiert wird.

Mit Hilfe der Beitrittszeitpunkte der Knoten der PROV Typen, kann die Anzahl der PROV Typen, die in einem GitLab2PROV Provenienzgraph enthalten sind, über den Zeitraum des Projektes des Provenienzgraphen dargestellt werden. In Abbildung 4.3 wird die Anzahl der Knoten der Provenienzgraphen der verwendeteten vier GitLab Projekte über die Zeiträume der Projekte hinweg dargestellt. Dafür wird ein Flächendiagramm verwendet, in welchem die Aufsummierung der einzelnen Flächen die Gesamtsumme der Knoten des Graphen repräsentiert.

Im Projekt ssa/BACARDI wächst die Anzahl der Knoten ab Februar 2018 stetig an. Dabei ist zu erkennen, dass ab Februar 2018 der Anteil der Activities an der Gesamtanzahl der Knoten des Provenienzgraphen des Projektes zunimmt. Die Anzahl der Entities pro Activity verändert sich ab Juli 2018 ebenfalls und nähert sich dem Endwert von ca. 2,566 zum Zeitpunkt der Generierung des Graphen an. Es lässt sich daher aussagen, dass im ssa/BACARDI Projekt ab Juli 2018 vermehrt Aktionen getätigt wurden, die im Provenienzgraphen pro Activity weniger Entities generieren als es im Zeitraum vor Juli 2018 der Fall war. Aktionen dieser Art sind z. B. Commits, die Änderungen an wenigen Dateien vornehmen, sowie jegliche Interaktionen, welche von Nutzern im Zusammenhang mit Issues und Merge Requests getätigt werden.

Zudem ist ein Unterschied im Anstieg der Knoten des Projektprovenienzgraphen ab Februar 2018 zu erkennen. Nach Februar 2018 ist der Anstieg der Anzahl der Knoten steiler als noch vor dem Februar 2018. Gegen Ende der Aufzeichnung, und damit kurz vor Generierung des Graphen im Juli 2020, flacht der Anstieg der Knotenanzahl ab. Bezogen auf die Projektaktivität kann anhand der Steigung ausgesagt werden, dass ab Februar 2018 generell mehr Aktionen im Projekt getätigt werden. Bis Anfang 2020 bleibt die Steigung, mit welcher die

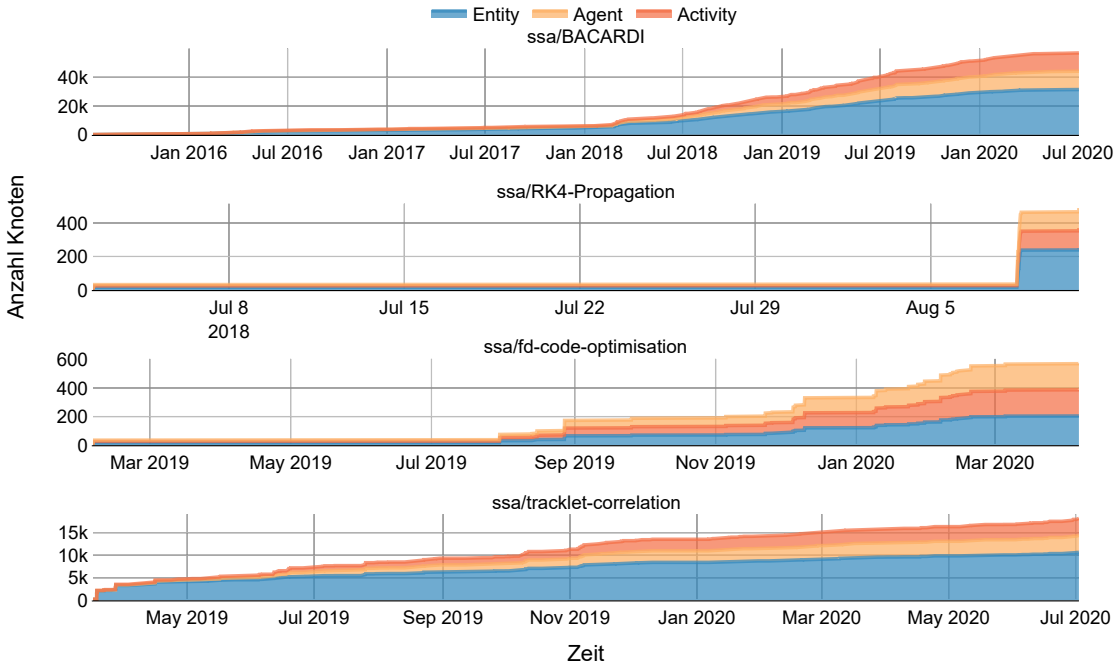


Abbildung 4.3: Anzahl der Knoten der PROV Typen über die Projekzeiträume

Anzahl der Knoten anwächst, ungefähr gleich, ab März 2020 lässt sich der Anfang eines Plateaus erkennen. Das Plateau und der damit zusammenhängende Rückgang der Anzahl der Aktionen, die im ssa/BACARDI Projekt getätigt werden und neue Knoten im Provenienzgraphen erzeugen, lässt sich auf den, durch die COVID-19-Pandemie bedingten, Minimalbetrieb des DLR zurückführen. Im Minimalbetrieb mussten die Mitarbeiter des DLR und damit auch die am BACARDI Projekt beteiligten Personen von der eigenen Wohnung aus arbeiten. Gewohnte Arbeitsprozesse wurden damit ausgehebelt. Zudem existierten für einige Personen Infrastrukturprobleme, sodass die Arbeit am Projekt nicht wie zuvor gewohnt möglich war.

In der Darstellung der Entwicklung der Knotenanzahl des Provenienzgraphen des GitLab Projektes ssa/RK4-Propagation, lässt sich ein klarer Anstieg an zwei Positionen auf der Zeitleiste der Darstellung erkennen: der erste direkt zu Beginn der Aufzeichnung, im Juli 2018, der zweite Anstieg findet im August 2018 statt. In beiden Fällen nimmt die Anzahl der Knoten des Graphen über einen kurzen Zeitraum zu. Zwischen, vor und nach den angesprochenen Zeiträumen stagniert das Wachstum der Anzahl der Knoten. Daran ist zu erkennen, dass sich die Entwicklungsaktivität des Projektes genau auf die beiden Zeiträume erstreckt, in denen die Anzahl der Knoten anwächst. Anhand der Plateaus und der Steigung des Graphen lässt sich erkennen, ob und wann ein Projekt aktiv weiterentwickelt wird.

Für das Projekt ssa/fd-code-optimization ist zu sehen, dass die ersten Knoten und damit die erste Entwicklungsaktivität im Februar 2019 stattfindet. Ab Juli 2019 nimmt das Wachstum der Anzahl der Knoten des Graphen deutlich zu und bleibt bis zum März 2020 stabil. Ab März 2020 lässt sich ein Plateau in der Anzahl der Knoten des Graphen erkennen.

Das Projekt ssa/tracklet-correlation verzeichnet die ersten Aktivitäten im März 2019. Bis zum Juli 2019 ist die Anzahl der Entities pro Activity hoch und nimmt im weiteren Verlauf des Graphen ab. Daher lässt sich vermuten, dass die Aktionen, die bis zum Juli 2019 im Projekt getätigt wurden, überwiegend Commits sind, welche Dateien hinzufügen oder verändern.

Im weiteren Verlauf des Graphen nähert sich die Anzahl der Entities pro Activity dem Wert an, der bei der Generierung des Graphen erreicht wird (ca. 3,082). Ähnlich zum ssa/BACARDI Projekt, kann aus dieser Beobachtung geschlossen werden, dass sich das Commit Verhalten der Entwickler verändert hat und weniger Dateien pro Commit verändert bzw. hinzugefügt werden, oder die Anzahl der Interaktionen von Entwickler mit GitLab Webresourcen wie Issues oder Merge Requests über die Zeit zugenommen hat, da diese im Provenienzgraphen ein geringere Anzahl an Entities pro Activity generieren.

Aus der Entwicklung der Knotenanzahl des Graphen lässt sich erkennen, ob in einem GitLab Projekt aktiv neue Aktionen stattfinden. Das Wachstum des Graphen stellt damit einen Indikator für die Entwicklungsaktivität eines Projektes dar. Aus der Veränderung der Anzahl der Entities pro Activity über den zeitlichen Verlauf eines Projektes, kann auf eine Verhaltensänderung bei Git Commits oder auf eine Zunahme von GitLab internen Ereignissen, wie z. B. Nutzerinteraktionen mit GitLab Webresourcen, geschlossen werden.

4.2.3 Prozessspezifische Fragestellungen

Die prozesspezifischen Fragestellungen werden, mit Ausnahme der ersten, ausschließlich anhand des Provenienzgraphen des GitLab Projektes ssa/BACARDI ausgewertet. Eine Auswertung für alle im Gesamtgraphen enthaltenen Projekte ist prinzipiell möglich, übersteigt jedoch den Rahmen dieser Arbeit.

Wer wirkt an welchen Projekten mit?

Eine Fragestellung, die das Management mehrerer Projekte betrifft, ist die Frage nach den Entwicklern, die an den Projekten beteiligt sind. Welche Projekte teilen sich Entwickler, und wie viele Entwickler sind an einem einzelnen Projekt beteiligt? In Abbildung 4.4 ist das Ergebnis der für diese Fragestellung entwickelten und in Abschnitt 3.4.3 vorgestellten Cypher Query in Form eines Sankey Diagrammes dargestellt.

Die Problematik, die es bei der Darstellung der Ergebnisse der Query zu lösen gilt, ist, wie abgebildet werden kann, an welchen Projekten die gleichen Agents beteiligt sind bzw. welche Projekte sich untereinander Agents teilen. Die Darstellung mit Hilfe eines Sankey Diagrammes wurde der Darstellung mit Hilfe eines Venn Diagrammes vorgezogen, da sich ein Venn Diagramm zwar gut für die Darstellung der Überschneidungen von zwei bis vier seperaten Mengen eignet, jedoch für mehr als vier Mengen unübersichtlich wird.

Auf der linken Seite der Abbildung werden die im Provenienzgraph enthaltenen GitLab Projekte dargestellt. Auf der rechten Seite werden die im Provenienzgraph enthaltenen PROV Agents dargestellt. Jeder der Agents bekommt eine Farbe zugewiesen, die in den folgenden Auswertungen zur Markierung des Agents wiederverwendet wird. Agents werden mit den Projekten verbunden, an denen sie beteiligt sind. Ein Agent ist genau dann an einem Projekt beteiligt, wenn der Agent im Gesamtprovenienzgraph mit einer Activity assoziiert wird, die dem Teilprovenienzgraphen des Projektes zugehört.

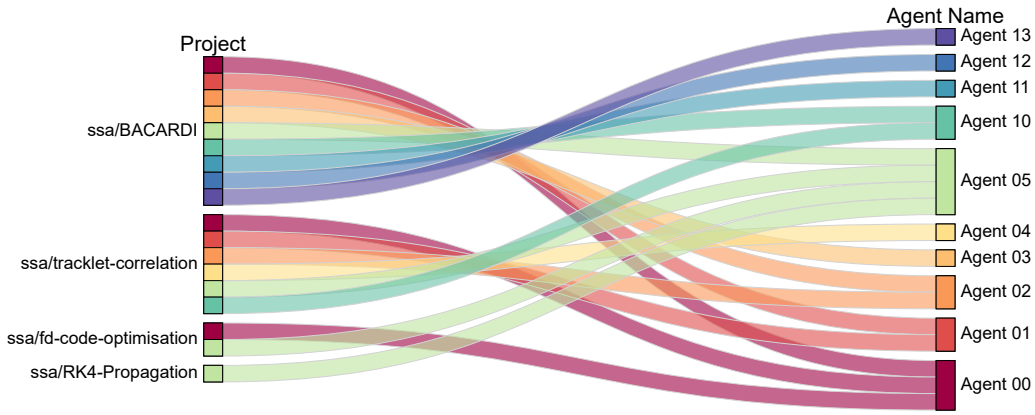


Abbildung 4.4: Agents pro Teilprojekt - Darstellung als Sankey Diagramm

Am GitLab Projekt ssa/BACARDI sind neun verschiedene Personen beteiligt, am Projekt ssa/tracklet-correlation sechs, am Projekt ssa/fd-code-optimisation zwei sowie am GitLab Projekt ssa/RK4-Propagation eine einzige Person. Agent 05 ist an allen vier der dargestellten Projekte beteiligt, Agent 00 an allen bis auf ssa/RK4-Propagation. An je zwei Projekten sind Agent 01, Agent 02 und Agent 10 beteiligt, alle drei Agents arbeiten an ssa/BACARDI sowie ssa/tracklet-correlation. Die restlichen Agents sind an einem Projekt beteiligt, bis auf Agent 04, der am ssa/tracklet-correlation Projekt mitarbeitet, gehören alle verbleibenden Agents dem ssa/BACARDI Projekt an.

Teilen sich zwei verschiedene Projekte mehrere Entwicklern, kann eine Verbindung zwischen den Projekten hergestellt werden, so z. B. zwischen den Projekten ssa/BACARDI und ssa/tracklet-correlation. Wie in der Vorstellung der GitLab Projekte erwähnt wurde, ist das Projektziel des GitLab Projektes ssa/tracklet-correlation, dass der entwickelte Programm Code von der BACARDI Software zur Korrelation von Objektbeobachtungsspuren verwendet wird. Dass die beiden Projekte sich Entwickler teilen, ist daher zu erwarten und dadurch zu erklären. Auch die anderen Projekte sind im Rahmen der DLR Software BACARDI gestartet worden, eine Verbindung über die gleichen Entwickler ist auch hier zu erwarten gewesen.

Wer ist wann in einem Projekt aktiv?

Die Entwicklung der Anzahl der aktiven Entwickler eines Projektes wird in diesem Abschnitt anhand des GitLab Projektes ssa/BACARDI demonstriert. In Abbildung 4.5 werden für jeden am GitLab Projekt ssa/BACARDI beteiligten Agent die Zeiträume dargestellt, in welchen der Agent aktiv war. Zur Darstellung wird ein Gantt Diagramm verwendet. Als aktiver Zeitraum eines Agents wird die Zeitperiode zwischen der ersten und letzten Activity definiert, die ein Agent in einem Projekt verantwortet bzw. mit welcher der Agent assoziiert wird.

Der aktive Zeitraum der Agents 00 und 05 beginnt direkt mit bzw. kurz nach Aufzeichnung der ersten Activities im Juli 2015. Beide Agents bleiben bis zum Zeitpunkt der Generierung des Provenienzgraphen des GitLab Projektes ssa/BACARDI aktiv. Ein halbes Jahr nach den ersten beiden Agents, beginnt der aktive Zeitraum des Agents 01 im Februar 2016. Agent 03 bleibt, wie die Agents 00 und 05, bis zum Zeitpunkt der Generierung des Graphen im BACARDI Projekt aktiv. Ungefähr anderthalb Jahre später, im Mai 2017, tätigt Agent 10 seine erste Aktion im GitLab Projekt ssa/BACARDI. Damit beginnt der aktive Zeitraum von Agent 10, welcher bis zum Ende der Aufzeichnungen aktiv am Projekt beteiligt ist. Im Mai 2018 und Juni 2018 treten zwei neue Agents dem Projekt bei: Agent 11 sowie Agent 02. Der aktive Zeitraum von Agent 02 endet drei Monate nach seinem Beginn im August 2018. Agent 11 tätigt im März 2019 seine letzte Aktion im ssa/BACARDI GitLab Projekt.

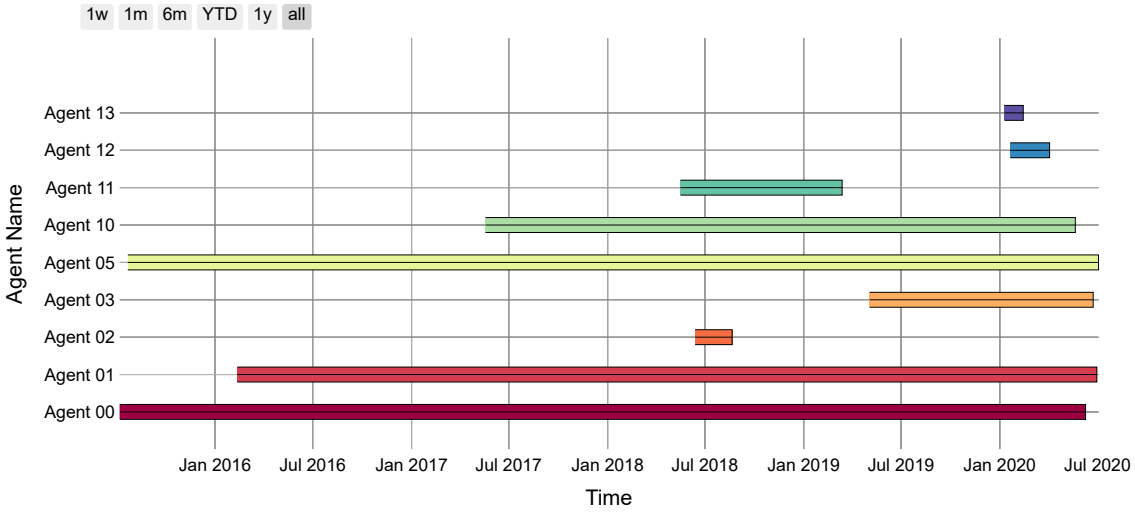


Abbildung 4.5: Aktivitätszeiträume der an den Projekten beteiligten Personen

Im Mai 2019 wird Agent 03 im BACARDI Projekt aktiv und bleibt es bis zum Ende der Aufzeichnungen. Im Januar 2020 werden erneut gleich zwei neue Agents aktiv, Agent 12 sowie Agent 13. Der Aktivitätszeitraum des Agent 13 erstreckt sich auf einen Monat und endet im Februar 2020. Agent 12 hingegen bleibt bis Anfang April 2020 aktiv.

Das BACARDI Projekt beginnt der Abbildung zufolge mit zwei Personen, die an der Entwicklung beteiligt sind. Über den Projektverlauf nimmt die Anzahl der beteiligten Personen zu. Seit Februar 2016 sind mindestens drei Personen an der Entwicklung des Projektes beteiligt, seit Mai 2017 mindestens vier und ab Mai 2019 mindestens fünf Personen. Hochphasen der Anzahl der zeitgleich am BACARDI GitLab Projekt beteiligten Personen sind im Zeitraum von Mai 2018 bis März 2019 mit sechs Personen und im Zeitraum von Januar 2020 bis April 2020 mit insgesamt bis zu sieben Personen zu erkennen.

Zeitleisten der Nutzerinteraktionen

Die Entwickler eines GitLab Projektes können zahlreiche Interaktionen mit dem Projekt tätigen. Wie im Methodikteil erläutert, zeichnet GitLab einen Großteil der Interaktionen in einer GitLab internen Eventrepräsentation auf. Alle Interaktionsevents werden wiederum in den GitLab2PROV Provenienzmodellen modelliert und sind im Provenienzgraph eines Projektes enthalten. Die Interaktionen geben unter anderem einen Überblick darüber, wie die Features, die GitLab bereitsstellt, von den Entwicklern eines Projektes verwendet werden, oder auch wie aktiv ein Projekt entwickelt wird.

In Abbildung 4.6 wird für jeden Interaktionseventtypen der Events des GitLab Projektes ssa/-BACARDI eine Zeitleiste dargestellt, auf der die Zeitpunkte, zu welchen bestimmte Events geschehen, eingezeichnet wurden. Die Zeitleisten der Eventtypen werden durch Abschnitte auf der X-Achse des Diagrammes in Halbjahresschritte aufgeteilt. Zur Beschreibung der Halbjahre werden Abkürzungen verwendet, so wird z. B. aus dem ersten Halbjahr des Jahres 2015 die Abkürzung 2015H1. Durch die Farben der Rauten wird dargestellt, welcher der Agents bzw. welcher der Entwickler das Event, welches die Raute markiert, verantwortet. Die Farben der Agents werden aus dem Diagramm in Abbildung 4.5 übernommen. Die Grundlage für die Darstellung bietet die in Quelltext Abbildung 3.6 dargestellte Cypher Query.

In die Auswertung des Diagrammes werden der Kürze halber nicht alle Eventtypen miteinbezogen. Bei der Auswertung wird der Fokus auf den im BACARDI Projekt eingeführten Entwicklungsprozess gelegt. Der Entwicklungsprozess wurde im Juli 2018 eingeführt, in der Auswertung werden die Zeitleisten daher in zwei Zeiträume eingeteilt, in den Zeitraum vor Einführung des Prozesses sowie in den Zeitraum nach Einführung des Prozesses.

Nutzerinteraktionen - Vor Prozesseinführung

Das erste Event, das im Diagramm und damit im Provenienzgraphen aufgezeichnet wird, ist ein von Agent 00 durchgeführter Commit und findet am 7. Juli 2015 statt. Der Verlauf der ersten drei dargestellten Halbjahre 2015H1, 2015H2 sowie 2016H1 ist miteinander vergleichbar. In den drei Halbjahren werden ausschließlich Commits getätigt. Die Commits werden in den ersten beiden Halbjahren von Agent 00 und Agent 05 verantwortet, im Halbjahr 2016H1 ist auch Agent 01 für Commits verantwortlich. Im Halbjahr 2016H2 nimmt der zeitliche Abstand zwischen Events zu, zum Jahreswechsel 2016/2017 lässt sich eine größere Pause zwischen Events feststellen.

Im Halbjahr 2017H1 nimmt die Anzahl der unterschiedlichen Eventtypen zu. Zum ersten Mal werden Issues und Merge Requests im Projekt erstellt bzw. geöffnet (*Issue Creation*, *Merge Request Creation*). Zum Ende des Halbjahres wird Agent 10 aktiv und merged zum ersten Mal im Projektverlauf eine Merge Request (*Merge*). Über den Verlauf des Halbjahres werden Issues geschlossen (*Close*), ein Issue wird nach Schließung erneut geöffnet (*Reopen*). Im Halbjahr 2017H1 werden unter anderem das erste Mal Kommentare verfasst (*Note*), das erste Mal Einträge einer ToDo-Liste abgehakt (*Mark Task As Done*), und es werden das erste Mal Issues bzw. Merge Request Entwickler zugewiesen (*Assign User*).

Im Halbjahr 2017H2 nimmt die Frequenz der Events im Vergleich zum vorherigen Halbjahr ab. Für die stattfindenden Events sind überwiegend Agent 00 sowie Agent 05 verantwortlicht. Im Vergleich zum Halbjahr 2017H1 werdem dem Projekt in regelmäßigeren Abständen Commits hinzugefügt. Zum Jahresende lässt sich erneut eine größere Pause zwischen Events feststellen.

Gegen Ende des Halbjahres 2018H1 wird der neue Entwicklungsprozess eingeführt. Bereits im Februar des Halbjahres ist zu beobachten, dass die Anzahl der verschiedenen Events, die im Projekt stattfinden, ansteigt. Interaktionsevents, die vorher semiregulär stattfanden, wie das Öffnen und Schließen von Issues (*Issue Creation* bzw. *Close*), finden in deutlich kürzeren Abständen und ohne größere Pausen zwischen Events des Typen statt. Die Anzahl und Frequenz anderer Events, die die Aktivität eines Issues wiederspiegeln, wie das Ändern der Issue Beschreibung (*Change Description*) und das Zuweisen des Issues an Entwickler, nimmt ebenfalls zu. Zusätzlich wird eine Vielzahl an Events zum ersten Mal oder zum ersten Mal erneut getätigt. Ein Beispiel dafür ist die Verwendung von sogenannten Meilensteinen an GitLab Webresourcen (*Change Milestone*). Diese können verwendet werden, um Issues bestimmten Meilensteinen der Projektplanung zuzuweisen. Im Halbjahr 2018H1 werden die aufgezeichneten Aktionen vorallem durch drei verschiedene Agents verantwortet: Agent 10, Agent 01 sowie Agent 00.

Vor dem Zeitpunkt der offiziellen Einführung des neuen Entwicklungsprozesses, ist das Halbjahr 2018H1 bereits das eventreichste Halbjahr der Aufzeichnungen seit Projektbeginn. Unter anderem ist dies dadurch erklärbar, dass einige der BACARDI Entwickler auch aktiv an der Entwicklung der DLR SE Guidelines beteiligt waren und im Verlauf der Entwicklung der Guidelines mehrere Unterpunkte der Guidelines bereits aktiv im BACARDI Projekt umsetzten, ohne einem formell definierten Entwicklungsprozess zu folgen.

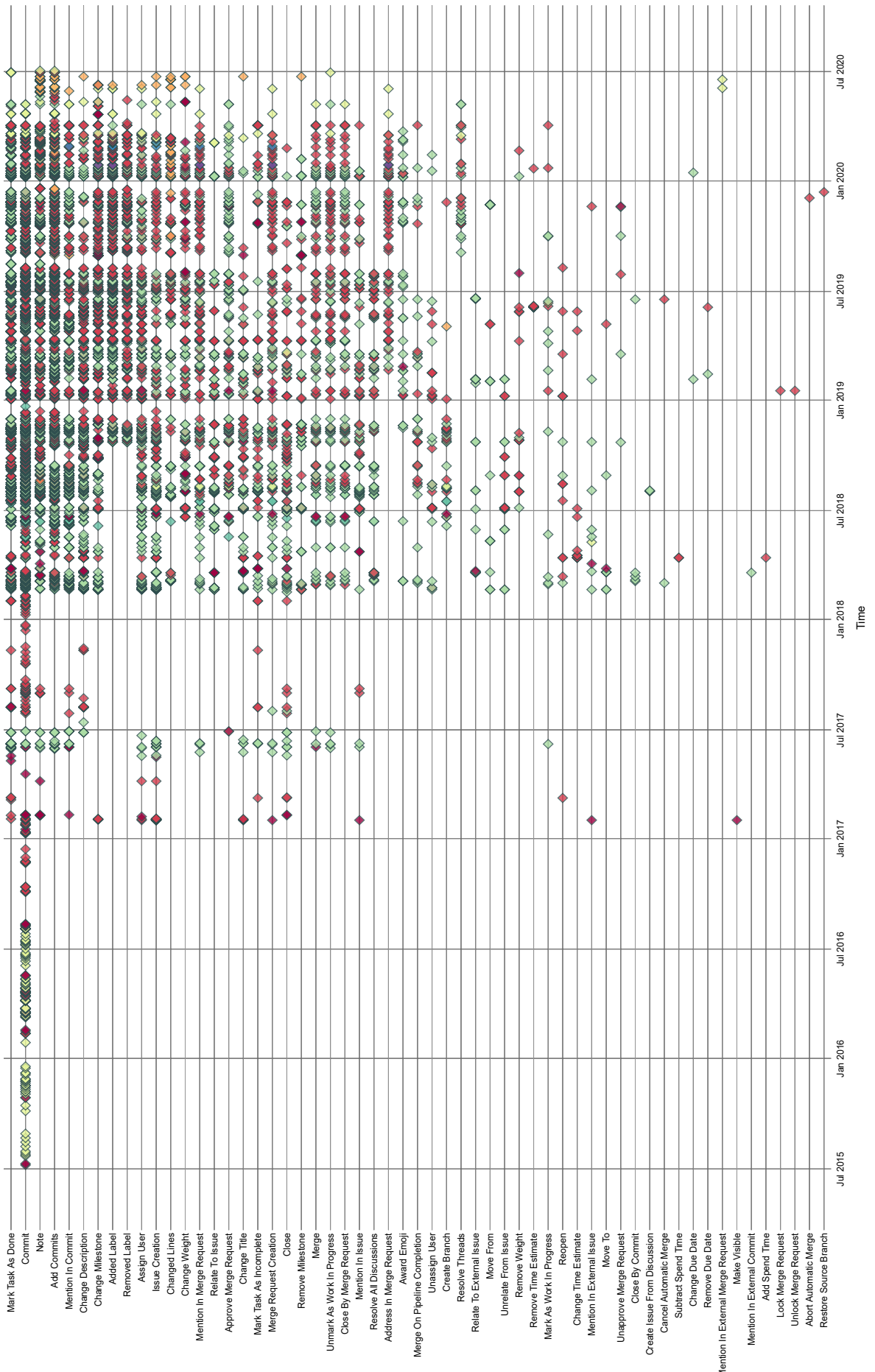


Abbildung 4.6: Zeitleisten der Interaktionsevents des GitLab Projektes ssa/BACARDI

Nutzerinteraktionen - Nach Prozesseinführung

Das Ende des Halbjahres 2018H1 markiert die Einführung des neuen Entwicklungsprozesses. Wie sich die in Abschnitt 4.1 vorgestellten Bestandteile des Entwicklungsprozesses im Interaktionsverhalten der Nutzer bzw. Entwickler des GitLab Projektes ssa/BACARDI wiederspiegeln, wird in den folgenden Absätzen erläutert.

Feature Entwicklung: Die Planung eines Features beginnt in einem Issue, welcher das geplante Feature beschreibt. Aus dem Issue heraus wird zu Beginn der Implementierung des Features ein neuer Git Branch eröffnet, in welchem das Feature entwickelt werden soll. Zeitgleich zur Erstellung des Branches wird eine Merge Request eröffnet, die den Feature Branch in den Baseline Branch, den Hauptbranch des Projektes, mergen soll. Die Events *Create Branch* sowie *Address in Merge Request* stellen das Öffnen eines Feature Branches aus einem GitLab Issue dar.

Werden Commits dem Feature Branch und damit der erstellten Merge Request hinzugefügt, wird dies im Event *Add Commits* festgehalten. Das Erstellen und das Hinzufügen von Commits zu Feature Branches, lässt sich anhand der genannten Eventtypen in den Nutzerinteraktionen des Projektes nach Einführung des neuen Entwicklungsprozesses feststellen.

Ist ein Feature weitgehend implementiert, kommt es zum Review. Im Review können einzelne Zeilen von Dateiänderungen angepasst werden (*Change Lines*) und Diskussionen über die vorgeschlagene Implementierung in den Kommentaren einer Merge Request geführt werden. Sind alle Punkte der Diskussionen erledigt, werden die Diskussionen als erledigt markiert (*Resolve Discussions*, *Resolve Threads*). Auch diese Events können nach Einführung des neuen Prozesses in den Zeitleisten der Nutzerinteraktionen gefunden werden und treten regelmäßig auf.

Sind alle Probleme rund um die Implementierung eines Features gelöst, kann der Feature Branch gemerged werden. Dazu muss die entsprechende Merge Request zunächst von einem Reviewer abgenommen werden (*Approve Merge Request*). Anschließend kann diese gemerged werden *Merge*. Beides lässt sich nach Einführung des neuen Prozesses beobachten. Nach Merge wird der Issue, in welchem das Feature geplant wurde, geschlossen. Das Schließen des Issues durch eine Merge Request wird durch Events des Typen *Close By Merge Request* aufgezeichnet.

Die Zeitleisten dDas Schließen des Issueer Event Typen sind abss lteigend nach der Anzahl der Events sortiert. Der Eventtyp der am häufigsten auftritt ist *Mark Task As Done*. Das Event wird beim Abhaken eines Eintrages einer TODO-Liste ausgelöst. TODO Listen können in GitLab durch das GitLab Flavoured Markdown in Textelementen auf Webresourcen wie Issues oder Merge Requests verwendet werden. Anhand der Anzahl der Events dieses Typen nach Einführung des neuen Prozesses, lässt sich vermuten, dass die in der Beschreibung des Entwicklungsprozesses genannte Definition of Done Checkliste in Form von TODO Listen auf einzelnen Webresourcen im GitLab Projekt verwendet wird.

Sprintplanung: Bei der Sprintplanung werden Gewichte an Issues vergeben, die den prognostizierten Zeitaufwand des Issues repräsentieren sollen. Die Vergabe von Gewichten sowie das Verändern oder Entfernen vergebener Gewichtswerte können auf den Zeitleisten der Eventtypen *Change Weight* und *Remove Weight* erkannt werden. Beide Eventtypen werden erst nach dem Zeitpunkt der offiziellen Einführung des Prozesses aufgezeichnet und werden anschließend regelmäßig verwendet.

Wird ein Issue für einen Sprint ausgewählt, wird dem Issue ein Meilenstein hinzugefügt, welcher den geplanten Sprint repräsentiert. Das Verändern eines Meilensteines eines Issues und das Entfernen eines Meilensteines wird durch die Events *Change Milestone* sowie *Remove Milestone* dargestellt. Beide Eventtypen werden erst nach Einführung des neuen Entwicklungsprozesses im Juli 2018 aktiv verwendet.

Beides sind Spuren der Vorgaben des neuen Prozesses und lassen darauf schließen, dass die Vorgaben des Prozesses umgesetzt wurden. Die Darstellung der Nutzerinteraktionen gibt nur einen Überblick über das GitLab Projekt. So ist noch nicht klar, ob die Aktionen für jeden neuen Issue getätigt werden, oder sich nur auf einen einzigen Issue konzentrieren. Jedoch können aus der Übersicht erste Indizien abgeleitet werden, die für die Umsetzung des neuen Prozesses sprechen.

GitLab API Veränderungen: Anhand der Eventzeitleisten, lassen sich Veränderungen der GitLab API feststellen. Die Events der Zeitleisten werden wie anfangs erklärt aus System Notes, einer GitLab internen Event Repräsentation, extrahiert. Verändert sich die bezeichnende Zeichenkette eines Events ist das in der Zeitleiste daran erkennbar, dass Events des alten Bezeichners ab dem API Update nicht mehr stattfinden und stattdessen Events des neuen Bezeichners aufgezeichnet werden. Dies ist z. B. der Fall für den Eventtypen *Resolve Discussions* der nach einem API Update in den Eventtypen *Resolve Threads* umbenannt wurde.

Weitere API Veränderungen lassen sich anhand der Eventtypen *Added Label* und *Removed Label* erkennen. Label können an Issues vergeben werden, um diese in bestimmte Kategorien aufzuteilen und anhand der Kategorien durchsuchbar zu machen. Im Diagramm werden die ersten Label Events in der zweiten Hälfte des Halbjahres 2018H2 verzeichnet. Das ist bereits deutlich nach der ersten Verwendung anderer Features, die im Zusammenhang mit Issues stehen. Nach Kontrolle per Hand lassen sich im GitLab Projekt bereits im Februar 2018 Label Events finden. Der Grund für den Unterschied zwischen den Aufzeichnungen im Provenienzgraph und dem Realzustand des Projektes ist eine Veränderung der Label Event API. Vor dem Zeitpunkt der Veränderung der API wurden Label Events ausschließlich von einer dedizierten Label Event API ausgegeben. Seit der API Anpassung, sind Label Events auch in System Notes zu finden. Da GitLab2PROV ausschließlich System Notes parsed, werden vor dem Zeitpunkt der API Anpassung keine Label Events aufgezeichnet.

Rollen der Entwickler: Die Rollenverteilung eines Projektes kann anhand der aufgezeichneten Nutzerinteraktionen untersucht werden. Gibt es eine Aufteilung von Entwicklern in Personen, die Code produzieren (“Coder”), und Entwicklern, die Code evaluieren (“Reviewer”), wird dies anhand der Nutzerinterkationen des Projektes deutlich. Neben den genannten Rollen der “Coder” und “Reviewer”, können auch weitere Rollen festgestellt werden. So lassen sich manche Aufgaben eher in dem Bereich des Projektmanagements zuordnen, wie z. B. das Verteilen von Issues an Entwickler oder speziell im BACARDI Projekt das Zuweisen von Gewichten an Issues im Rahmen der Sprint Planung. Eine Person, die hauptsächlich für diese Aufgaben verantwortlich ist, wird wahrscheinlich auch in dieser Rolle arbeiten.

Bei der Darstellung der Nutzerinteraktionen des GitLab Projektes ssa/BACARDI lassen sich mehrere Beobachtungen feststellen. Beispielsweise ist nach Einführung des neuen Entwicklungsprozesses ab Juli 2018 festzustellen, dass das Abnehmen von Merge Requests (*Approve Merge Request*) hauptsächlich von drei Akteuren vorgenommen wird, von Agent 10, Agent 00 sowie Agent 01. Das Hinzufügen von Commits zum Git Repository des Projektes hingegen wird von mehr als den drei genannten Agents verantwortet. Es lässt sich dementsprechend

vermuten, dass die Verantwortung des Reviews der Veränderungen, die durch eine Merge Request vorgenommen werden, bei den Agents 00, 01 sowie 10 liegt, die Verantwortung des Code Produzierens sich aber auf eine größere Entwicklergruppe erstreckt.

Entwicklungspausen und Urlaubszeiten: Im Diagramm sind Lücken zwischen Events erkennbar, die in allen Zeitleisten des Diagrammes zum gleichen Zeitpunkt stattfinden. Gerade in Halbjahren, die ansonsten eine Vielzahl an Events beinhalten, fallen diese Lücken auf.

Ein Beispiel einer solchen Lücke ist der Jahreswechsel vom Jahr 2018 auf das Jahr 2019. Durch alle Events hinweg werden in diesem Zeitraum keine Aktionen am GitLab Projekt getätigt, die als Interaktionsevents aufgezeichnet werden. Ein zweites Beispiel einer solchen Lücke zu einer anderen Jahreszeit ist im August bzw. September 2019 zu erkennen. Auch im Jahreswechsel von 2019 auf 2020 lässt sich eine Lücke zwischen Events beobachten.

Die Lücken im Verlauf der Interaktionsevents lassen sich durch typische Urlaubszeiten und Feiertage erklären. In den ersten Halbjahren des Projektes ließen sich Lücken zum Jahresende bereits erkennen. Durch die allgemein niedrigere Frequenz der Events in den ersten Jahren der Entwicklung, sind Lücken zu Urlaubs- und Ferienzeiten nicht sehr gut zu erkennen.

Corona und weitere Veränderungen: Als Reaktion auf die, durch die COVID-19-Pandemie bedingte, allgemeine Gesundheitsgefahr, wurde am 19. März 2020 im DLR der Krisenfall ausgerufen und der sogenannte Minimalbetrieb begonnen. Alle Mitarbeitenden, die nicht zur Aufreichterhaltung kritischer Vor-Ort-Prozesse im DLR notwending sind arbeiten seit diesem Zeitpunkt von zuhause aus. Diese Mitarbeitenden haben während des Zeitraums des Minimalbetriebs keinen Zutritt zu Standorten des DLR.

Der Minimalbetrieb kann als ein Grund dafür festgehalten werden, dass ab März 2020 die Frequenz der aufgezeichneten Events abnimmt und sich damit der Abstand zwischen den einzelnen Events vergrößert. Zudem kann die Pandemie auch ein Grund dafür sein, dass einzelne Akteure nicht mehr so aktiv am Projekt mitarbeiten, wie dies noch vor Beginn des Minimalbetriebs der Fall war. Durch mangelnden Infrastrukturausbau in ländlichen Regionen kann es schwierig sein auf die DLR interne Infrastruktur über VPN Lösungen zuzugreifen, da die notwendige Bandbreite und Stabilität der vorhandenen Internetverbindungen nicht zwingend gegeben sind.

Neben den Auswirkungen der COVID-19-Pandemie sind auch Veränderungen an der Aktivität einzelner Entwickler erkennbar. Agent 10, Agent 00 sowie Agent 01 waren in den Jahren von 2017 bis 2020 für den Großteil der im Provenienzgraph aufgezeichneten Ereignisse verantwortlich. Ab April 2020 nimmt die Regelmäßigkeit mit welcher Agent 10 und Agent 01 Events auslösen ab. Dieser Umstand lässt sich durch die Versetzung der beiden Entwickelnden, die hinter den Pseudonymen Agent 10 und Agent 01 stehen, zu einem weiteren Softwareprojekt erklären.

Mit diesen Betrachtungen schließt das Evaluationskapitel.

5 Fazit

Ziel dieser Bachelorarbeit war es einen Ansatz zu entwickeln, der es ermöglicht die Entwicklungsprozesse von GitLab Projekten basierend auf den Provenienzgraphen der Projekte zu untersuchen.

Dazu wurden die Provenienzgraphen verschiedener GitLab Projekte mit dem Python Tool GitLab2PROV generiert. Zur Analyse der Graphen wurden die Graphen in die Graphdatenbank Neo4j importiert. Die Analyse wurde mit eigens entwickelten Abfragen der Graph Query Language Cypher durchgeführt, die Ergebnisse der Abfragen wurde mit Hilfe der Visualisierungslibrary Plotly dargestellt.

Aus den Eigenschaften der Graphstruktur des Provenienzgraphen eines GitLab Projektes ließen sich Aussagen über die Ereignisse treffen, die in den GitLab Projekten stattgefunden haben. Dabei konnte aus der Anzahl der Knoten und Kanten eines Provenienzgraphen auf die, bei der Generierung der Graphen verwendeten, Provenienzmodelle geschlossen werden und somit auf die im Graphen abgebildeten Aktionen.

Für die Graphstruktur von GitLab2PROV Provenienzgraphen wurde ein zeitliches Wachstumskriterium entwickelt, anhand dessen das Wachstum des Provenienzgraphen eines GitLab Projektes über den zeitlichen Verlauf des Projektes dargestellt werden kann. Anhand der Steigung des Wachstums kann abgelesen werden, wie aktiv an einem Projekt entwickelt wird bzw. wie viele neue Knoten dem Provenienzgraphen des Projektes hinzugefügt werden. Zudem konnten, anhand der Veränderung der Graphstruktur über den zeitlichen Verlauf, Indizien abgeleitet werden, die auf Veränderung des Nutzungsverhaltens des GitLab Projektes hindeuten.

Für die Entwickler eines Projektes wurden Aktivitätszeiträume definiert anhand welcher erkannt werden kann, wie sich die Anzahl der Entwickler verändert, die zeitgleich an einem Projekt arbeiten.

Anhand der Zeitleisten der Nutzerinteraktionen eines GitLab Projektes, können Aussagen über die Rollen einzelner Entwickler, die Projektaktivität sowie das genutzte GitLab Feature Set getätigt werden. Im Zusammenhang mit der Einführung neuer Entwicklungsprozesse, kann anhand der Nutzerinteraktionen untersucht werden, wie sich der neue Prozess auf die Interaktionen der Entwickler mit dem GitLab Projekt auswirkt. Am Beispiel der DLR Software BACARDI wurde dies demonstriert.

Anhand der Provenienzgraphen eines Projektes, kann wie in der Arbeit präsentiert noninvasiv in die Entwicklungsprozesse hinter den Projekten Einsicht genommen werden. Mit

zusätzlichem spezialisierten Domainenwissen, wie der formalen Spezifizierung von Entwicklungsprozessen, kann anhand des Provenienzgraphen eines Projektes untersucht werden, inwieweit sich Spuren des Entwicklungsprozesses im Projekt wiederfinden lassen.

Ausblick

Der in der Arbeit verwendete Ansatz zur Analyse von GitLab Projekten bietet eine Vielzahl an Erweiterungsmöglichkeiten angefangen bei der Generierung der Provenienzgraphen. Noch werden nicht alle Features, die GitLab bereitstellt, von den GitLab2PROV Provenienzmodellen abgedeckt. Abgesehen davon unterstützt GitLab2PROV noch keine vollständige Rückwärtskompatibilität mit älteren GitLab API Versionen. Gerade Features, wie Continous Integration Pipelines, die GitLab unter anderem zur Einrichtung automatisierter Test Suites und Code Checks anbietet, sind eine vielversprechende Erweiterungsmöglichkeit der verwendeten Provenienzmodelle.

Neben der Übersicht über den Prozess hinter einem GitLab Projekt, sind für Themenbereiche wie der Prozessoptimierung und der Einschätzung der Projekt Performance auch stets quantifizierbare Metriken gefragt. Einige dieser Metriken, wie z. B. die “Time-to-Close” für Issues, existieren bereits für Softwareprojekte, die mit Hilfe von Issue Trackern oder Code Hosting Plattformen wie GitLab und GitHub entwickelt werden. Die Entwicklung provenienzbasierter, quantifizierbarer Metriken ist ein mögliches Thema zukünftiger Arbeiten.

Komplexe Softwaresysteme können aus mehreren Teilprojekten bestehen. Um Zusammenhänge, Verbindungen und Überschneidungen der verschiedenen Teilprojekte zu untersuchen, kann der in dieser Arbeit vorgestellte Ansatz verwendet werden, um die Interkonnektivität der Projekte zu analysieren. In dieser Arbeit wurde gezeigt, dass ausgehend von den Provenienzgraphen mehrerer Projekte bereits beantwortet werden kann, welche Projekte sich Entwickler teilen. GitLab Projekte unterstützen Events, die Projekte miteinander verbinden können. Die Analyse der Projektverbindungen bleibt als offene Frage für zukünftige Arbeiten.

Literaturverzeichnis

[BC20] Jesús Barrasa und Adam Cowley. *neosemantics (n10s) : Neo4j RDF amp; Semantics toolkit*. [Online; accessed 2020-07-14]. Mai 2020. url: https://neo4j.com/labs/neosemantics-rdf/.

[Bec+14] David Beckett u. a. „RDF 1.1 Turtle“. In: *World Wide Web Consortium* (2014).

[Bie+20] Stefan Bieliauskas u. a. *DLR-SC/prov-db-connector: v0.4.0*. Version v0.4.0. März 2020. doi: 10.5281/zenodo.3732584. url: https://doi.org/10.5281/zenodo.3732584.

[Boe20] Claas de Boer. *cdboer/ba-thesis-jupyter-notebook: BA submission release*. Version v1.0. Aug. 2020. doi: 10.5281/zenodo.3970207. url: https://doi.org/10.5281/zenodo.3970207.

[BS20] Claas de Boer und Andreas Schreiber. *DLR-SC/gitlab2prov: Second release*. Version v0.2. Aug. 2020. doi: 10.5281/zenodo.3969434. url: https://doi.org/10.5281/zenodo.3969434.

[CCM09] Sérgio Manuel Serra da Cruz, Maria Luiza M Campos und Marta Mattoso. „Towards a taxonomy of provenance in scientific workflow management systems“. In: *2009 Congress on Services-I*. IEEE. 2009, S. 259–266.

[Din+10] Li Ding u. a. „Reflections on provenance ontology encodings“. In: *International Provenance and Annotation Workshop*. Springer. 2010, S. 198–205.

[Fra+18] Nadime Francis u. a. „Cypher: An evolving query language for property graphs“. In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, S. 1433–1445.

[FR91] Thomas MJ Fruchterman und Edward M Reingold. „Graph drawing by force-directed placement“. In: *Software: Practice and experience* 21.11 (1991), S. 1129–1164.

[Git20] GitLab. *GitLab - The First Single Application For The Entire Devops Lifecycle*. 2020. url: https://about.gitlab.com/ (besucht am 28. 06. 2020).

[GQL20] GQL. *Graph Query Language GQL*. 2020. url: https://www.gqlstandards.org/ (besucht am 28. 06. 2020).

[GM13] Paul Groth und Luc Moreau. *PROV-Overview*. W3C Note. W3C, Apr. 2013. url: http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/.

[Gro13] W3C SPARQL Working Group. *SPARQL 1.1 Overview*. W3C Recommendation. W3C, März 2013. url: https://www.w3.org/TR/sparql11-overview/ (besucht am 26. 06. 2020).

[Has08] Ahmed E Hassan. „The road ahead for mining software repositories“. In: *2008 Frontiers of Software Maintenance*. IEEE. Nov. 2008, S. 48–57. doi: 10.1109/FOSM.2008.4659248.

[HS20] Carina Haupt und Tobias Schlauch. *DLR Software Engineering Initiative*. 2020. url: https://rse.dlr.de/ (besucht am 28. 06. 2020).

[HSM18] Carina Haupt, Tobias Schlauch und Michael Meinel. „The Software Engineering Initiative of DLR - Overcome the obstacles and develop sustainable software". In: *2018 ACM/IEEE International Workshop on Software Engineering for Science*. Juni 2018. url: https://elib.dlr.de/120462/.

[Huy18] Trung Dong Huynh. *prov - A library for W3C Provenance Data Model supporting PROV-JSON, PROV-XML and PROV-O (RDF)*. [Online; accessed 2020-06-29]. 20. Nov. 2018. url: https://github.com/trungdong/prov.

[Imp18] openCypher Implementers Group. *Cypher Query Language Reference (Version 9)*. 2018. url: https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf.

[Inc20] Neo4j Inc. *Neo4j Graph Platform - The Leader in Graph Databases*. 2020. url: https://neo4j.com/ (besucht am 28. 06. 2020).

[Kim19] Nikolay Kim. *aiohttp - Async http client/server framework*. [Online; accessed 2020-06-29]. 9. Okt. 2019. url: https://github.com/aio-libs/aiohttp.

[Mor10] Luc Moreau. „The Foundations for Provenance on the Web". In: *Found. Trends Web Sci.* 2.2–3 (Feb. 2010), S. 99–241. issn: 1555-077X. doi: 10.1561/1800000010. url: https://doi.org/10.1561/1800000010.

[Mor+13a] Luc Moreau u. a. *PROV-DM: The PROV Data Model*. Hrsg. von Luc Moreau und Paolo Missier. Apr. 2013. url: http://www.w3.org/TR/2013/REC-prov-dm-20130430/.

[Mor+13b] Luc Moreau u. a. „PROV-N: the provenance notation". In: *W3C Recommendation* (2013).

[Nie+13] Tom De Nies u. a. „Git2PROV: Exposing Version Control System Content as W3C PROV". In: *Proceedings of the ISWC 2013 Posters & Demonstrations Track, Sydney, Australia, October 23, 2013*. 2013, S. 125–128. url: http://ceur-ws.org/Vol-1035/iswc2013%5C_demo%5C_32.pdf.

[OOB18] Wellington Oliveira, Daniel De Oliveira und Vanessa Braganholo. „Provenance Analytics for Workflow-Based Computational Experiments: A Survey". In: *ACM Comput. Surv.* 51.3 (Mai 2018). issn: 0360-0300. doi: 10.1145/3184900. url: https://doi.org/10.1145/3184900.

[PCC19] Heather S. Packer, Adriane Chapman und Leslie Carr. „GitHub2PROV: Provenance for Supporting Software Project Management". In: *11th International Workshop on Theory and Practice of Provenance (TaPP 2019)*. Philadelphia, PA: USENIX Association, Juni 2019. url: https://www.usenix.org/conference/tapp2019/presentation/packer.

[RWE15] Ian Robinson, Jim Webber und Emil Eifrem. *Graph databases: new opportunities for connected data*. Ö'Reilly Media, Inc.", 2015.

[RN10] Marko A Rodriguez und Peter Neubauer. „Constructions from dots and lines". In: *Bulletin of the American Society for Information Science and Technology* 36.6 (2010), S. 35–41.

[Ryd18] David Reinsel–John Gantz–John Rydning. „The digitization of the world from edge to core". In: *Framingham: International Data Corporation* (2018).

[SMH18] Tobias Schlauch, Michael Meinel und Carina Haupt. *DLR Software Engineering Guidelines*. Techn. Ber. Aug. 2018. url: https://elib.dlr.de/121502/.

[SPG05] Yogesh L Simmhan, Beth Plale und Dennis Gannon. „A survey of data provenance in e-science". In: *ACM Sigmod Record* 34.3 (2005), S. 31–36.

[Sma19] Nigel Small. *py2neo*. [Online; accessed 2020-07-14]. 10. Mai 2019. url: http://py2neo.org/.

[Sto+19] Martin Stoffers u. a. „BACARDI: A System to track Space Debris“. In: *ESA NEO and DEBRIS DETECTION CONFERENCE - EXPLOITING SYNERGIES* -. Feb. 2019. url: https://elib.dlr.de/126572/.

[Vic+10] Chad Vicknair u. a. „A comparison of a graph database and a relational database: a data provenance perspective“. In: *Proceedings of the 48th annual Southeast regional conference*. 2010, S. 1–6.

[VK14] Denny Vrandečić und Markus Krötzsch. „Wikidata: a free collaborative knowledgebase“. In: *Communications of the ACM* 57.10 (2014), S. 78–85.

[WKS10] Heinrich Wendel, Markus Kunde und Andreas Schreiber. „Provenance of Software Development Processes“. In: *Provenance and Annotation of Data and Processes - Third International Provenance and Annotation Workshop, IPAW 2010, Troy, NY, USA, June 15-16, 2010. Revised Selected Papers*. 2010, S. 59–63. doi: 10.1007/978-3-642-17819-1_7. url: https://doi.org/10.1007/978-3-642-17819-1%5C_7.

[WLC14] David Wood, Markus Lanthaler und Richard Cyganiak. *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation. W3C, Feb. 2014. url: https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ (besucht am 23. 06. 2020).